Fundamentals of Artificial Intelligence
Chapter 03: **Problem Solving as Search**

Roberto Sebastiani

DISI, Università di Trento, Italy – roberto.sebastiani@unitn.it
http://disi.unitn.it/rseba/DIDATTICA/fai_2020/

Teaching assistant: Mauro Dragoni – dragoni@fbk.eu
http://www.maurodragoni.com/teaching/fai/

M.S. Course "Artificial Intelligence Systems", academic year 2020-2021

Last update: Tuesday 8th December, 2020, 13:05

Outline

1. Problem-Solving Agents
2. Example Problems
3. Search Generalities
4. Uninformed Search Strategies
   - Breadth-First Search
   - Uniform-cost Search
   - Depth-First Search
   - Depth-Limited Search & Iterative Deepening
5. Informed Search Strategies
   - Greedy Search
   - A^* Search
   - Heuristic Functions

Outline

Problem Solving as Search

One of the dominant approaches to AI problem solving:
formulate a problem/task as search in a state space.

Main Paradigm

1. Goal formulation: define the successful world states
   - Ex: a set of states, a Boolean test function ...
2. Problem formulation:
   - define a representation for states
   - define legal actions and transition functions
3. Search: find a solution by means of a search process
   - solutions are sequences of actions
4. Execution: given the solution, perform the actions

⟹ Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving:
formulate a problem/task as search in a state space.

Main Paradigm

1. Goal formulation: define the successful world states
   - Ex: a set of states, a Boolean test function ...
2. Problem formulation:
   - define a representation for states
   - define legal actions and transition functions
3. Search: find a solution by means of a search process
   - solutions are sequences of actions
4. Execution: given the solution, perform the actions

⟹ Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving:
formulate a problem/task as search in a state space.

Main Paradigm

1. Goal formulation: define the successful world states
   - Ex: a set of states, a Boolean test function ...
2. Problem formulation:
   - define a representation for states
   - define legal actions and transition functions
3. Search: find a solution by means of a search process
   - solutions are sequences of actions
4. Execution: given the solution, perform the actions

$\Longrightarrow$ Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving:
formulate a problem/task as search in a state space.

Main Paradigm

1. Goal formulation: define the successful world states
   - Ex: a set of states, a Boolean test function ...
2. Problem formulation:
   - define a representation for states
   - define legal actions and transition functions
3. Search: find a solution by means of a search process
   - solutions are sequences of actions
4. Execution: given the solution, perform the actions

$\Longrightarrow$ Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving:
formulate a problem/task as search in a state space.

Main Paradigm

1. Goal formulation: define the successful world states
   - Ex: a set of states, a Boolean test function ...
2. Problem formulation:
   - define a representation for states
   - define legal actions and transition functions
3. Search: find a solution by means of a search process
   - solutions are sequences of actions
4. Execution: given the solution, perform the actions

⟹ Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving:
formulate a problem/task as search in a state space.

Main Paradigm

1. Goal formulation: define the successful world states
   - Ex: a set of states, a Boolean test function ...
2. Problem formulation:
   - define a representation for states
   - define legal actions and transition functions
3. Search: find a solution by means of a search process
   - solutions are sequences of actions
4. Execution: given the solution, perform the actions

$\Longrightarrow$ Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search

One of the dominant approaches to AI problem solving:
formulate a problem/task as search in a state space.

Main Paradigm

1. Goal formulation: define the successful world states
   - Ex: a set of states, a Boolean test function ...
2. Problem formulation:
   - define a representation for states
   - define legal actions and transition functions
3. Search: find a solution by means of a search process
   - solutions are sequences of actions
4. Execution: given the solution, perform the actions

$\Longrightarrow$ Problem-solving agents are (a kind of) goal-based agents

Problem Solving as Search: Example

Example: Traveling in Romania

- Informal description: On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest
- Formulate goal: (Be in) Bucharest
- Formulate problem:
  - States: various cities
  - Actions: drive between cities
  - Initial state: Arad
- Search for a solution: sequence of cities from Arad to Bucharest
  - e.g. Arad, Sibiu, Fagaras, Bucharest
  - explore a search tree/graph

Note

The agent is assumed to have no heuristic knowledge about traveling in Romania to exploit.

Problem Solving as Search: Example

Example: Traveling in Romania

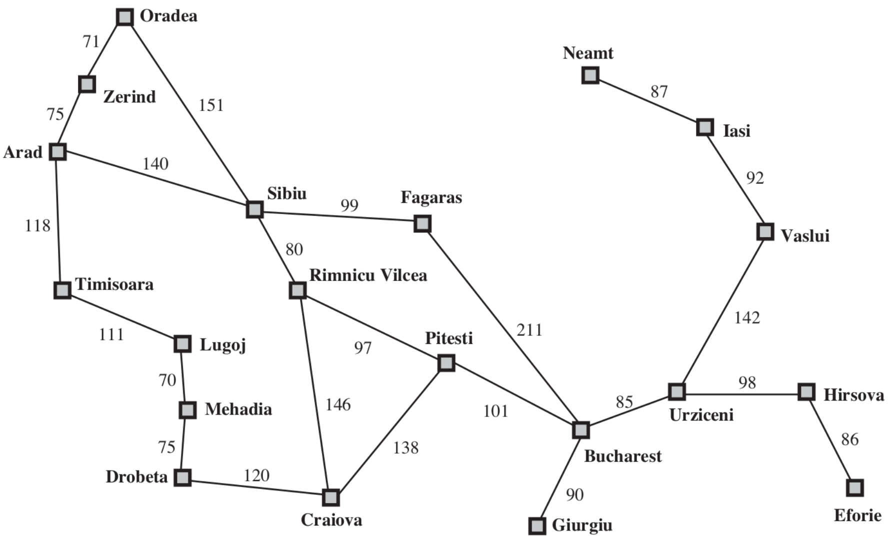
- Informal description: On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest
- Formulate goal: (Be in) Bucharest
- Formulate problem:
  - States: various cities
  - Actions: drive between cities
  - Initial state: Arad
- Search for a solution: sequence of cities from Arad to Bucharest
  - e.g. Arad, Sibiu, Fagaras, Bucharest
  - explore a search tree/graph

Note

The agent is assumed to have no heuristic knowledge about traveling in Romania to exploit.

Problem Solving as Search: Example [cont.]

A simplified road map of part of Romania.



(© S. Russell & P. Norwig, AIMA)

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter only)

- state representations are atomic
  - $\Longrightarrow$ world states are considered as wholes, with no internal structure
  - Ex: Arad, Sibiu, Zerind, Bucharest,...
- the environment is observable
  - $\Longrightarrow$ the agent always knows the current state
  - Ex: Romanian cities & roads have signs
- the environment is discrete
  - $\Longrightarrow$ at any state there are only finitely many actions to choose from
  - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)
- the environment is known
  - $\Longrightarrow$ the agent knows which states are reached by each action
  - ex: the agent has the map
- the environment is deterministic
  - $\Longrightarrow$ each action has exactly one outcome
  - Ex: from Arad choose go to Sibiu $\Longrightarrow$ next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter only)

- state representations are atomic
  - $\Longrightarrow$ world states are considered as wholes, with no internal structure
  - Ex: Arad, Sibiu, Zerind, Bucharest,...
- the environment is observable
  - $\Longrightarrow$ the agent always knows the current state
  - Ex: Romanian cities & roads have signs
- the environment is discrete
  - $\Longrightarrow$ at any state there are only finitely many actions to choose from
  - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)
- the environment is known
  - $\Longrightarrow$ the agent knows which states are reached by each action
  - ex: the agent has the map
- the environment is deterministic
  - $\Longrightarrow$ each action has exactly one outcome
  - Ex: from Arad choose go to Sibiu $\Longrightarrow$ next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter only)

- state representations are atomic
  - $\Longrightarrow$ world states are considered as wholes, with no internal structure
  - Ex: Arad, Sibiu, Zerind, Bucharest,...
- the environment is observable
  - $\Longrightarrow$ the agent always knows the current state
  - Ex: Romanian cities & roads have signs
- the environment is discrete
  - $\Longrightarrow$ at any state there are only finitely many actions to choose from
  - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)
- the environment is known
  - $\Longrightarrow$ the agent knows which states are reached by each action
  - ex: the agent has the map
- the environment is deterministic
  - $\Longrightarrow$ each action has exactly one outcome
  - Ex: from Arad choose go to Sibiu $\Longrightarrow$ next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter only)

- state representations are atomic
  - $\Longrightarrow$ world states are considered as wholes, with no internal structure
  - Ex: Arad, Sibiu, Zerind, Bucharest,...
- the environment is observable
  - $\Longrightarrow$ the agent always knows the current state
  - Ex: Romanian cities & roads have signs
- the environment is discrete
  - $\Longrightarrow$ at any state there are only finitely many actions to choose from
  - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)
- the environment is known
  - $\Longrightarrow$ the agent knows which states are reached by each action
  - ex: the agent has the map
- the environment is deterministic
  - $\Longrightarrow$ each action has exactly one outcome
  - Ex: from Arad choose go to Sibiu $\Longrightarrow$ next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter only)

- state representations are atomic
  - $\Longrightarrow$ world states are considered as wholes, with no internal structure
  - Ex: Arad, Sibiu, Zerind, Bucharest,...
- the environment is observable
  - $\Longrightarrow$ the agent always knows the current state
  - Ex: Romanian cities & roads have signs
- the environment is discrete
  - $\Longrightarrow$ at any state there are only finitely many actions to choose from
  - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)
- the environment is known
  - $\Longrightarrow$ the agent knows which states are reached by each action
  - ex: the agent has the map
- the environment is deterministic
  - $\Longrightarrow$ each action has exactly one outcome
  - Ex: from Arad choose go to Sibiu $\Longrightarrow$ next step in Sibiu

Problem Solving as Search [cont.]

Assumptions for Problem-solving Agents (this chapter only)

- state representations are atomic

  $\Longrightarrow$ world states are considered as wholes, with no internal structure
  - Ex: Arad, Sibiu, Zerind, Bucharest,...
- the environment is observable

  $\Longrightarrow$ the agent always knows the current state
  - Ex: Romanian cities & roads have signs
- the environment is discrete

  $\Longrightarrow$ at any state there are only finitely many actions to choose from
  - Ex: from Arad, (go to) Sibiu, or Zerind, or Timisoara (see map)
- the environment is known

  $\Longrightarrow$ the agent knows which states are reached by each action
  - ex: the agent has the map
- the environment is deterministic

  $\Longrightarrow$ each action has exactly one outcome
  - Ex: from Arad choose go to Sibiu $\Longrightarrow$ next step in Sibiu

Problem Solving as Search [cont.]

Notes about search

- Search happens inside the agent
  - a planning stage before acting
  - different from searching in the world
- An agent is given a description of what to achieve, not an algorithm to solve it

  $\Longrightarrow$ the only possibility is to search for a solution
- Searching can be computationally very demanding (NP-hard)
- Can be driven with benefits by knowledge of the problem (heuristic knowledge) $\Longrightarrow$ informed/heuristic search

Problem Solving as Search [cont.]

Notes about search

- Search happens inside the agent
  - a planning stage before acting
  - different from searching in the world
- An agent is given a description of what to achieve, not an algorithm to solve it
  $\Longrightarrow$ the only possibility is to search for a solution
- Searching can be computationally very demanding (NP-hard)
- Can be driven with benefits by knowledge of the problem (heuristic knowledge) $\Longrightarrow$ informed/heuristic search

Problem Solving as Search [cont.]

Notes about search

- Search happens inside the agent
  - a planning stage before acting
  - different from searching in the world
- An agent is given a description of what to achieve, not an algorithm to solve it

  $\Longrightarrow$ the only possibility is to search for a solution
- Searching can be computationally very demanding (NP-hard)
- Can be driven with benefits by knowledge of the problem (heuristic knowledge) $\Longrightarrow$ informed/heuristic search

Problem Solving as Search [cont.]

Notes about search

- Search happens inside the agent
  - a planning stage before acting
  - different from searching in the world
- An agent is given a description of what to achieve, not an algorithm to solve it

  $\Longrightarrow$ the only possibility is to search for a solution
- Searching can be computationally very demanding (NP-hard)
- Can be driven with benefits by knowledge of the problem (heuristic knowledge) $\Longrightarrow$ informed/heuristic search

Problem Solving as Search [cont.]

Notes about search

- Search happens inside the agent
  - a planning stage before acting
  - different from searching in the world
- An agent is given a description of what to achieve, not an algorithm to solve it

  $\Longrightarrow$ the only possibility is to search for a solution
- Searching can be computationally very demanding (NP-hard)
- Can be driven with benefits by knowledge of the problem (heuristic knowledge) $\Longrightarrow$ informed/heuristic search

Problem-solving Agent: Schema

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```


While executing the solution sequence the agent ignores its percepts when choosing an action since it knows in advance what they will be ("open loop system")

Problem-solving Agent: Schema

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
     if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```


While executing the solution sequence the agent ignores its percepts when choosing an action since it knows in advance what they will be ("open loop system")

Well-defined problems and solutions

Problem Formulation: Components

- the initial state the agent starts in
  - Ex: *In*(*Arad*)
- the set of applicable actions available in a state (ACTIONS(S))
  - Ex: if *s* is *In*(*Arad*), then the applicable actions are {*Go*(*Sibiu*), *Go*(*Timisoara*), *Go*(*Zerind*)}
- a description of what each action does (aka transition model)
  - RESULT(S,A): state resulting from applying action A in state S
  - Ex: RESULT(IN(ARAD), GO(ZERIND)) is IN(ZERIND)
- the goal test determining if a given state is a goal state
  - Explicit (e.g.: {*In*(*Bucharest*)})
  - Implicit (e.g. (Ex: CHECKMATE(X))
- the path cost function assigns a numeric cost to each path
  - in this chapter: the sum of the costs of the actions along the path

Well-defined problems and solutions

Problem Formulation: Components

- the initial state the agent starts in
  - Ex: $In(Arad)$
- the set of applicable actions available in a state (ACTIONS(S))
  - Ex: if s is $In(Arad)$, then the applicable actions are $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$
- a description of what each action does (aka transition model)
  - RESULT(S,A): state resulting from applying action A in state S
  - Ex: RESULT(IN(ARAD), GO(ZERIND)) is IN(ZERIND)
- the goal test determining if a given state is a goal state
  - Explicit (e.g.: $\{In(Bucharest)\}$)
  - Implicit (e.g. (Ex: CHECKMATE(X))
- the path cost function assigns a numeric cost to each path
  - in this chapter: the sum of the costs of the actions along the path

Well-defined problems and solutions

Problem Formulation: Components

- the initial state the agent starts in
  - Ex: $In(Arad)$
- the set of applicable actions available in a state (ACTIONS(S))
  - Ex: if s is $In(Arad)$, then the applicable actions are $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$
- a description of what each action does (aka transition model)
  - RESULT(S,A): state resulting from applying action A in state S
  - Ex: RESULT(IN(ARAD), GO(ZERIND)) is IN(ZERIND)
- the goal test determining if a given state is a goal state
  - Explicit (e.g.: $\{In(Bucharest)\}$)
  - Implicit (e.g. (Ex: CHECKMATE(X))
- the path cost function assigns a numeric cost to each path
  - in this chapter: the sum of the costs of the actions along the path

Well-defined problems and solutions

Problem Formulation: Components

- the initial state the agent starts in
  - Ex: *In*(*Arad*)
- the set of applicable actions available in a state (ACTIONS(S))
  - Ex: if *s* is *In*(*Arad*), then the applicable actions are {*Go*(*Sibiu*), *Go*(*Timisoara*), *Go*(*Zerind*)}
- a description of what each action does (aka transition model)
  - RESULT(S,A): state resulting from applying action A in state S
  - Ex: RESULT(IN(ARAD), GO(ZERIND)) is IN(ZERIND)
- the goal test determining if a given state is a goal state
  - Explicit (e.g.: {*In*(*Bucharest*)})
  - Implicit (e.g. (Ex: CHECKMATE(X))
- the path cost function assigns a numeric cost to each path
  - in this chapter: the sum of the costs of the actions along the path

Well-defined problems and solutions

Problem Formulation: Components

- the initial state the agent starts in
  - Ex: *In*(*Arad*)
- the set of applicable actions available in a state (ACTIONS(S))
  - Ex: if *s* is *In*(*Arad*), then the applicable actions are {*Go*(*Sibiu*), *Go*(*Timisoara*), *Go*(*Zerind*)}
- a description of what each action does (aka transition model)
  - RESULT(S,A): state resulting from applying action A in state S
  - Ex: RESULT(IN(ARAD), GO(ZERIND)) is IN(ZERIND)
- the goal test determining if a given state is a goal state
  - Explicit (e.g.: {*In*(*Bucharest*)})
  - Implicit (e.g. (Ex: CHECKMATE(X))
- the path cost function assigns a numeric cost to each path
  - in this chapter: the sum of the costs of the actions along the path

Well-defined problems and solutions

Problem Formulation: Components

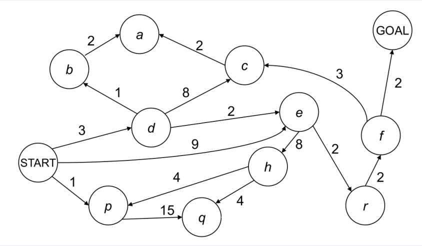
- the initial state the agent starts in
  - Ex: *In*(*Arad*)
- the set of applicable actions available in a state (ACTIONS(S))
  - Ex: if *s* is *In*(*Arad*), then the applicable actions are {*Go*(*Sibiu*), *Go*(*Timisoara*), *Go*(*Zerind*)}
- a description of what each action does (aka transition model)
  - RESULT(S,A): state resulting from applying action A in state S
  - Ex: RESULT(IN(ARAD), GO(ZERIND)) is IN(ZERIND)
- the goal test determining if a given state is a goal state
  - Explicit (e.g.: {*In*(*Bucharest*)})
  - Implicit (e.g. (Ex: CHECKMATE(X))
- the path cost function assigns a numeric cost to each path
  - in this chapter: the sum of the costs of the actions along the path

Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

- the state space forms a directed graph (e.g. the Romania map)
  - typically too big to be created explicitly and be stored in full
  - in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state
- an optimal solution is a solution with the lowest path cost

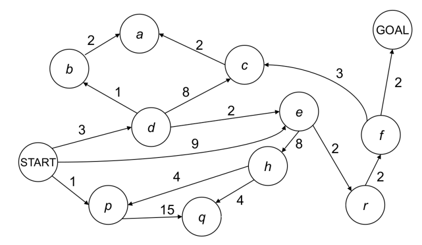


Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

- the state space forms a directed graph (e.g. the Romania map)
  - typically too big to be created explicitly and be stored in full
  - in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state
- an optimal solution is a solution with the lowest path cost



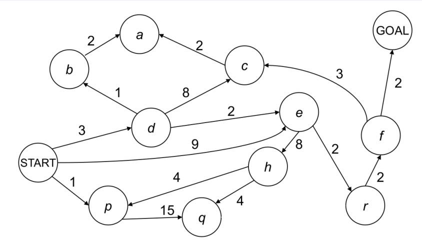
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

- the state space forms a directed graph (e.g. the Romania map)
  - typically too big to be created explicitly and be stored in full
  - in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state
- an optimal solution is a solution with the lowest path cost

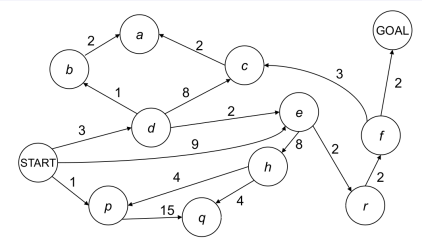


Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

- the state space forms a directed graph (e.g. the Romania map)
  - typically too big to be created explicitly and be stored in full
  - in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state
- an optimal solution is a solution with the lowest path cost

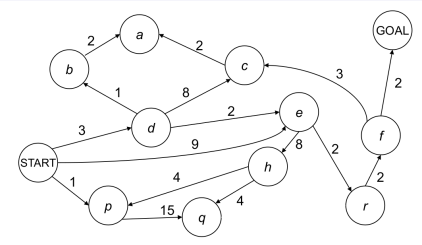


Well-defined problems and solutions [cont.]

State Space, Graphs, Paths, Solutions and Optimal Solutions

Initial state, actions, and transition model implicitly define the state space of the problem

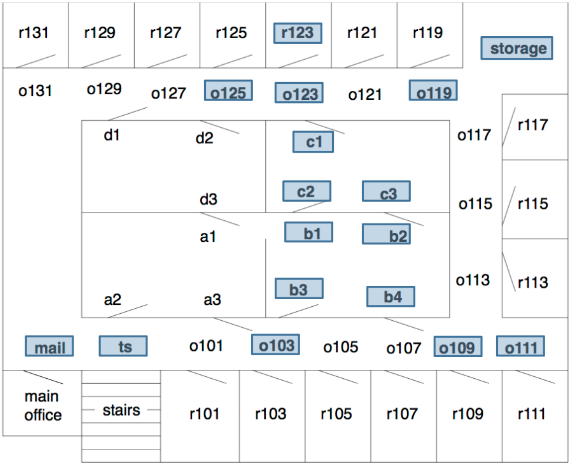
- the state space forms a directed graph (e.g. the Romania map)
  - typically too big to be created explicitly and be stored in full
  - in a state space graph, each state occurs only once
- a path is a sequence of states connected by actions
- a solution is a path from the initial state to a goal state
- an optimal solution is a solution with the lowest path cost



Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
- Initial state
- Goal state
- State graph
- Optimal solution

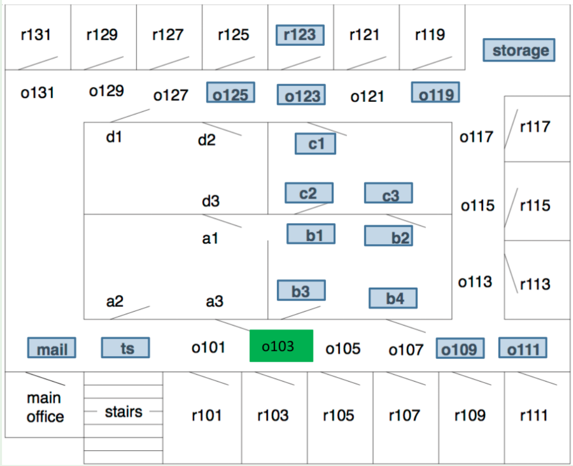


(Courtesy of Maria Simi, UniPI)

Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
- Initial state
- Goal state
- State graph
- Optimal solution

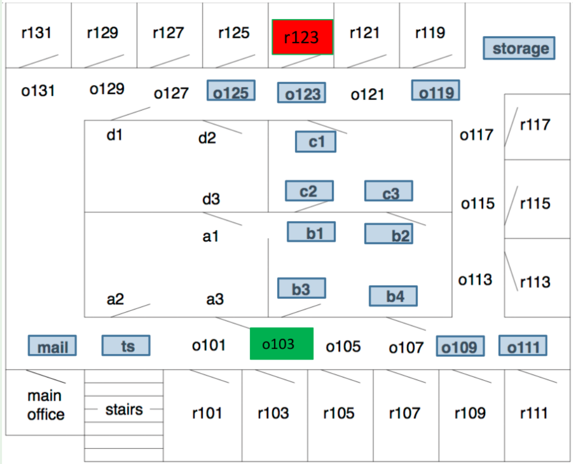


(Courtesy of Maria Simi, UniPI)

Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
- Initial state
- Goal state
- State graph
- Optimal solution

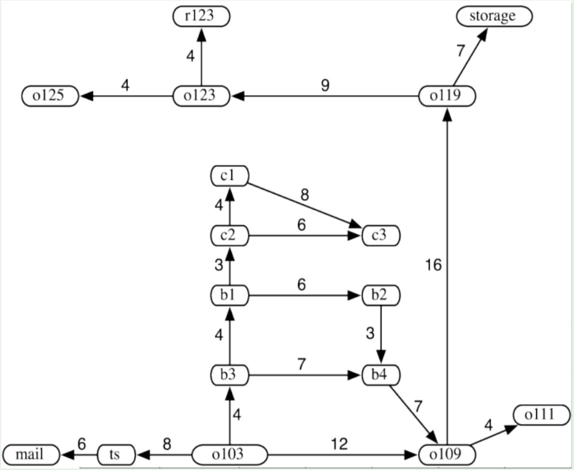


(Courtesy of Maria Simi, UniPI)

Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
- Initial state
- Goal state
- State graph
- Optimal solution

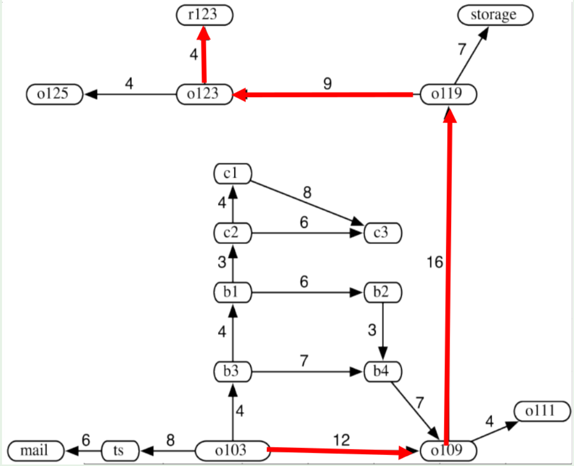


(Courtesy of Maria Simi, UniPI)

Example: Path finding for a Delivery Robot

Task: move from o103 to r123

- States
- Initial state
- Goal state
- State graph
- Optimal solution



(Courtesy of Maria Simi, UniPI)

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are models of reality (i.e. abstract descriptions)

- real world is absurdly complex
  $\Longrightarrow$ state space must be abstracted for problem solving
- lots of details removed because irrelevant to the problem
  - Ex: exact position, “turn steering wheel to the left by 20 degree”, ...
- abstraction: the process of removing detail from representations
  - abstract state represents many real states
  - abstract action represents complex combination of real actions
- valid abstraction: can expand any abstract solution into a solution in the detailed world
- useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

The choice of a good abstraction involves removing as much detail as possible, while retaining validity and ensuring that the abstract actions are easy to carry out.

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are models of reality (i.e. abstract descriptions)

- real world is absurdly complex
  $\Longrightarrow$ state space must be abstracted for problem solving
- lots of details removed because irrelevant to the problem
  - Ex: exact position, "turn steering wheel to the left by 20 degree", ...
- abstraction: the process of removing detail from representations
  - abstract state represents many real states
  - abstract action represents complex combination of real actions
- valid abstraction: can expand any abstract solution into a solution in the detailed world
- useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

The choice of a good abstraction involves removing as much detail as possible, while retaining validity and ensuring that the abstract actions are easy to carry out.

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are models of reality (i.e. abstract descriptions)

- real world is absurdly complex
  $\Longrightarrow$ state space must be abstracted for problem solving
- lots of details removed because irrelevant to the problem
  - Ex: exact position, "turn steering wheel to the left by 20 degree", ...
- abstraction: the process of removing detail from representations
  - abstract state represents many real states
  - abstract action represents complex combination of real actions
- valid abstraction: can expand any abstract solution into a solution in the detailed world
- useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

The choice of a good abstraction involves removing as much detail as possible, while retaining validity and ensuring that the abstract actions are easy to carry out.

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are models of reality (i.e. abstract descriptions)

- real world is absurdly complex
  $\Longrightarrow$ state space must be abstracted for problem solving
- lots of details removed because irrelevant to the problem
  - Ex: exact position, "turn steering wheel to the left by 20 degree", ...
- abstraction: the process of removing detail from representations
  - abstract state represents many real states
  - abstract action represents complex combination of real actions
- valid abstraction: can expand any abstract solution into a solution in the detailed world
- useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

The choice of a good abstraction involves removing as much detail as possible, while retaining validity and ensuring that the abstract actions are easy to carry out.

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are models of reality (i.e. abstract descriptions)

- real world is absurdly complex
  $\Longrightarrow$ state space must be abstracted for problem solving
- lots of details removed because irrelevant to the problem
  - Ex: exact position, "turn steering wheel to the left by 20 degree", ...
- abstraction: the process of removing detail from representations
  - abstract state represents many real states
  - abstract action represents complex combination of real actions
- valid abstraction: can expand any abstract solution into a solution in the detailed world
- useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

The choice of a good abstraction involves removing as much detail as possible, while retaining validity and ensuring that the abstract actions are easy to carry out.

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are models of reality (i.e. abstract descriptions)

- real world is absurdly complex
  $\Longrightarrow$ state space must be abstracted for problem solving
- lots of details removed because irrelevant to the problem
  - Ex: exact position, "turn steering wheel to the left by 20 degree", ...
- abstraction: the process of removing detail from representations
  - abstract state represents many real states
  - abstract action represents complex combination of real actions
- valid abstraction: can expand any abstract solution into a solution in the detailed world
- useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

The choice of a good abstraction involves removing as much detail as possible, while retaining validity and ensuring that the abstract actions are easy to carry out.

Well-defined problems and solutions [cont.]

Abstraction

Problem formulations are models of reality (i.e. abstract descriptions)

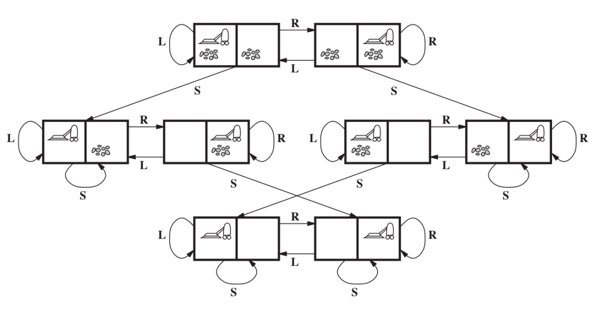
- real world is absurdly complex
  $\Longrightarrow$ state space must be abstracted for problem solving
- lots of details removed because irrelevant to the problem
  - Ex: exact position, “turn steering wheel to the left by 20 degree”, ...
- abstraction: the process of removing detail from representations
  - abstract state represents many real states
  - abstract action represents complex combination of real actions
- valid abstraction: can expand any abstract solution into a solution in the detailed world
- useful abstraction: if carrying out each of the actions in the solution is easier than in the original problem

The choice of a good abstraction involves removing as much detail as possible, while retaining validity and ensuring that the abstract actions are easy to carry out.

Outline

Toy Example: Simple Vacuum Cleaner

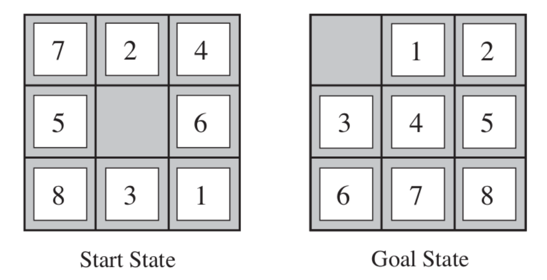
- States: 2 locations, each $\{clean, dirty\}$: $2 \cdot 2^2 = 8$ states
- Initial State: any
- Actions: $\{Left, Right, Suck\}$
- Transition Model: Left [Right] if A [B], Suck if clean $\Longrightarrow$ no effect
- Goal Test: check if squares are clean
- Path Cost: each step costs 1 $\Longrightarrow$ path cost is # of steps in path



(© S. Russell & P. Norwig, AIMA)

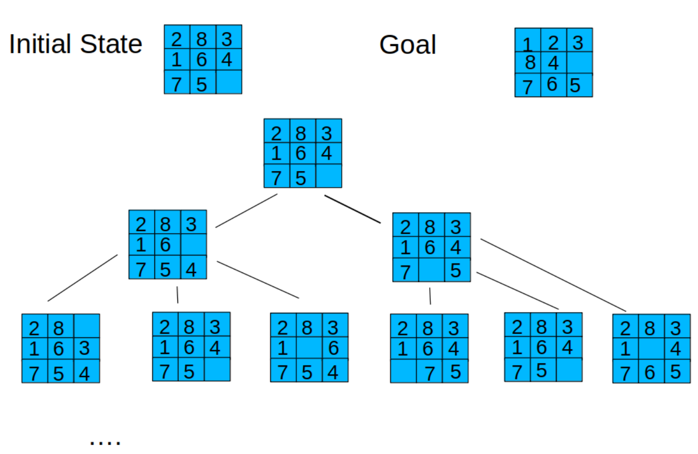
Toy Example: The 8-Puzzle

- States: Integer location of each tile $\Longrightarrow$ $9!/2$ reachable states
- Initial State: any
- Actions: moving $\{Left, Right, Up, Down\}$ the empty space
- Transition Model: empty switched with the tile in target location
- Goal Test: checks state corresponds with goal configuration
- Path Cost: each step costs 1 $\Longrightarrow$ path cost is # of steps in path



(© S. Russell & P. Norwig, AIMA)

Toy Example: The 8-Puzzle [cont.]

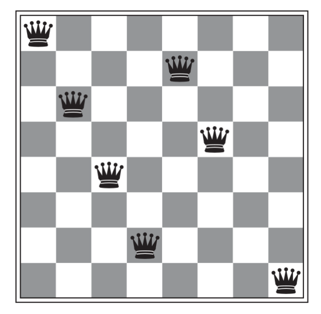


(Courtesy of Michela Milano, UniBO)

NP-complete: N-Puzzle ($N = k^2 - 1$): $N!/2$ reachable states

Toy Example: 8-Queens Problem

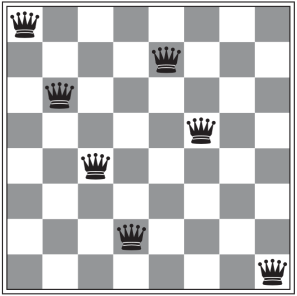
- States: any arrangement of 0 to 8 queens on the board
  $\Longrightarrow$ $64 \cdot 63 \cdot ... \cdot 57 \approx 1.8 \cdot 10^{14}$ possible sequences
- Initial State: no queens on the board
- Actions: add a queen to any empty square
- Transition Model: returns the board with a queen added
- Goal Test: 8 queens on the board, none attacked by other queen
- Path Cost: none



(© S. Russell & P. Norwig, AIMA)

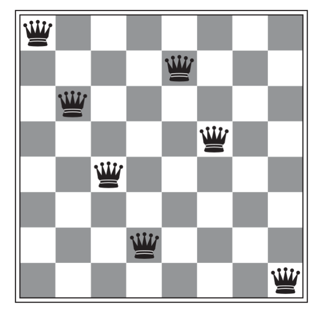
Toy Example: 8-Queens Problem

- States: any arrangement of 0 to 8 queens on the board
  $\Longrightarrow$ $64 \cdot 63 \cdot ... \cdot 57 \approx 1.8 \cdot 10^{14}$ possible sequences
- Initial State: no queens on the board
- Actions: add a queen to any empty square
- Transition Model: returns the board with a queen added
- Goal Test: 8 queens on the board, none attacked by other queen
- Path Cost: none



Toy Example: 8-Queens Problem (incremental)

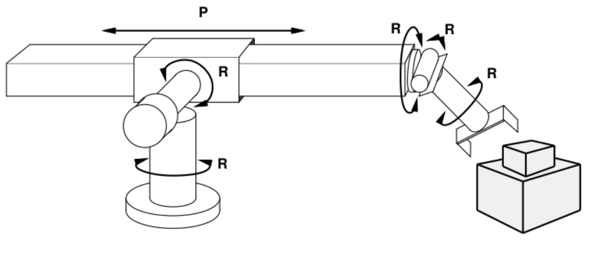
- States: $n \leq 8$ queens on board, one per column in the n leftmost columns, no queen attacking another
  $\Longrightarrow$ 2057 possible sequences
- Actions: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
- ...



(© S. Russell & P. Norwig, AIMA)

Real-World Example: Robotic Assembly

- States: real-valued coordinates of robot joint angles, and of parts of the object to be assembled
- Initial State: any arm position and object configuration
- Actions: continuous motions of robot joints
- Transition Model: position resulting from motion
- Goal Test: complete assembly (without robot)
- Path Cost: time to execute



(© S. Russell & P. Norwig, AIMA)

Other Real-World Examples

- Airline travel planning problems
- Touring problems
- VLSI layout problem
- Robot navigation
- Automatic assembly sequencing
- Protein design
- ...

Outline

1. Problem-Solving Agents
2. Example Problems
3. Search Generalities
4. Uninformed Search Strategies
   - Breadth-First Search
   - Uniform-cost Search
   - Depth-First Search
   - Depth-Limited Search & Iterative Deepening
5. Informed Search Strategies
   - Greedy Search
   - A^* Search
   - Heuristic Functions

Searching for Solutions

Search: Generate sequences of actions.

- Expansion: one starts from a state, and applying the operators (or successor function) will generate new states
- Search strategy: at each step, choose which state to expand.
- Search Tree: It represents the expansion of all states starting from the initial state (the root of the tree)
- The leaves of the tree represent either:
  - states to expand
  - solutions
  - dead-ends

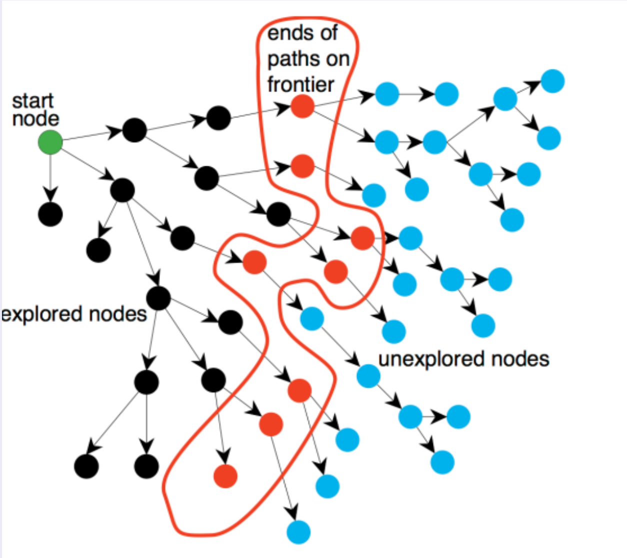
Tree Search Algorithms

Tree Search: Basic idea

- Off-line, simulated exploration of state space
  - start from initial state
  - pick one leaf node, and generate its successors (a.k.a. expanding a node) states)
    - set of current leaves called frontier (a.k.a. fringe, open list)
    - strategy for picking leaves critical (search strategy)
  - ends when either a goal state is reached, or no more candidates to expand are available (or time-out/memory-out occur)

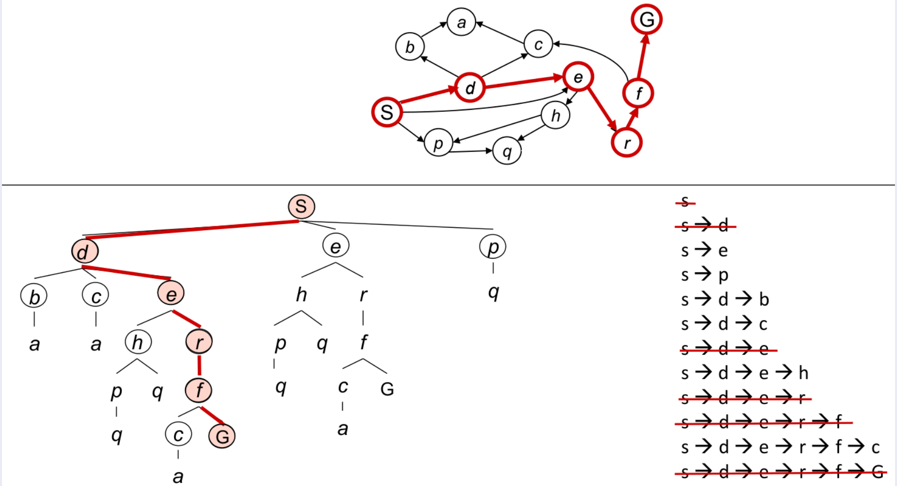
function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

Tree Search Algorithms [cont.]



(Courtesy of Maria Simi, UniPI)

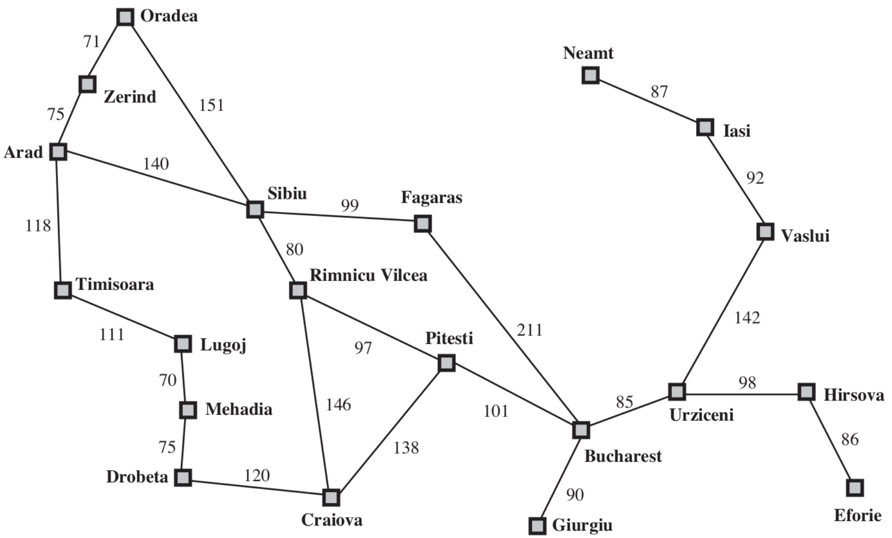
Tree-Search Example



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Tree-Search Example: Trip from Arad to Bucharest

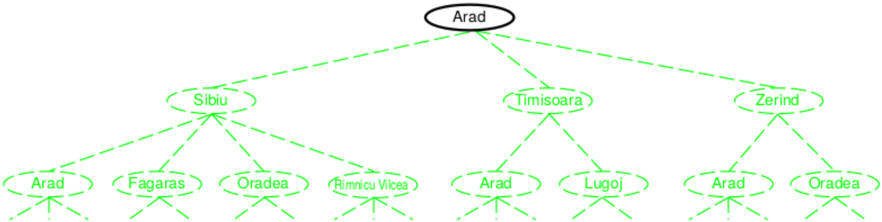
A simplified road map of part of Romania.



(© S. Russell & P. Norwig, AIMA)

Expanding the search tree

- Initial state: $\{Arad\}$
- Expand initial state $\Longrightarrow$ $\{Sibiu, Timisoara, Zerind\}$
- Pick&expand Sibiu $\Longrightarrow$ $\{Arad, Fagaras, Oradea, Rimnicu\ Vicea\}$
- ...

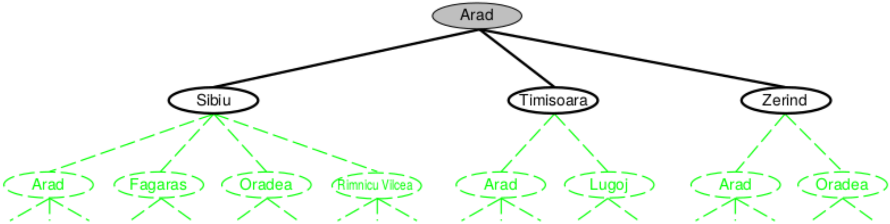


Beware: Arad $\mapsto$ Sibiu $\mapsto$ Arad (repeated state $\Longrightarrow$ loopy path!)

Tree-Search Example: Trip from Arad to Bucharest

Expanding the search tree

- Initial state: $\{Arad\}$
- Expand initial state $\Longrightarrow$ $\{Sibiu, Timisoara, Zerind\}$
- Pick&expand Sibiu $\Longrightarrow$ $\{Arad, Fagaras, Oradea, Rimnicu\ Vicea\}$
- ...



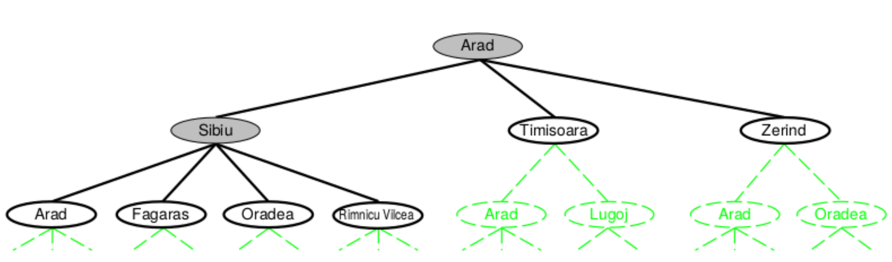
(© S. Russell & P. Norwig, AIMA)

Beware: Arad $\mapsto$ Sibiu $\mapsto$ Arad (repeated state $\Longrightarrow$ loopy path!)

Tree-Search Example: Trip from Arad to Bucharest

Expanding the search tree

- Initial state: $\{Arad\}$
- Expand initial state $\Longrightarrow$ $\{Sibiu, Timisoara, Zerind\}$
- Pick&expand Sibiu $\Longrightarrow$ $\{Arad, Fagaras, Oradea, Rimnicu\ Vicea\}$
- ...



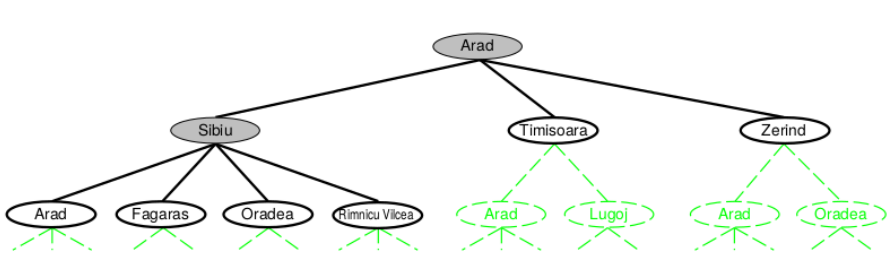
(© S. Russell & P. Norwig, AIMA)

Beware: Arad $\mapsto$ Sibiu $\mapsto$ Arad (repeated state $\Longrightarrow$ loopy path!)

Tree-Search Example: Trip from Arad to Bucharest

Expanding the search tree

- Initial state: $\{Arad\}$
- Expand initial state $\Longrightarrow \{Sibiu, Timisoara, Zerind\}$
- Pick&expand Sibiu $\Longrightarrow \{Arad, Fagaras, Oradea, Rimnicu\ Vicea\}$
- ...



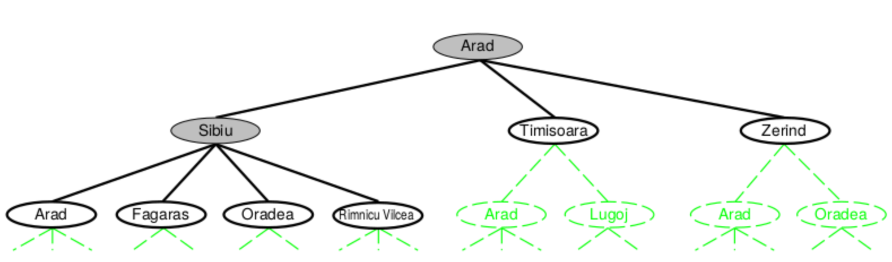
(© S. Russell & P. Norwig, AIMA)

Beware: Arad $\mapsto$ Sibiu $\mapsto$ Arad (repeated state $\Longrightarrow$ loopy path!)

Tree-Search Example: Trip from Arad to Bucharest

Expanding the search tree

- Initial state: $\{Arad\}$
- Expand initial state $\Longrightarrow$ $\{Sibiu, Timisoara, Zerind\}$
- Pick&expand Sibiu $\Longrightarrow$ $\{Arad, Fagaras, Oradea, Rimnicu\ Vicea\}$
- ...



(© S. Russell & P. Norwig, AIMA)

Beware: Arad $\mapsto$ Sibiu $\mapsto$ Arad (repeated state $\Longrightarrow$ loopy path!)

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another
  $\Longrightarrow$ same state explored more than once
- Failure to detect repeated states can:
  - cause infinite loops
  - turn linear problem into exponential

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another

  $\Longrightarrow$ same state explored more than once
- Failure to detect repeated states can:
  - cause infinite loops
  - turn linear problem into exponential

(© S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another

  $\Longrightarrow$ same state explored more than once
- Failure to detect repeated states can:
  - cause infinite loops
  - turn linear problem into exponential

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another

  $\Longrightarrow$ same state explored more than once

- Failure to detect repeated states can:
  - cause infinite loops
  - turn linear problem into exponential

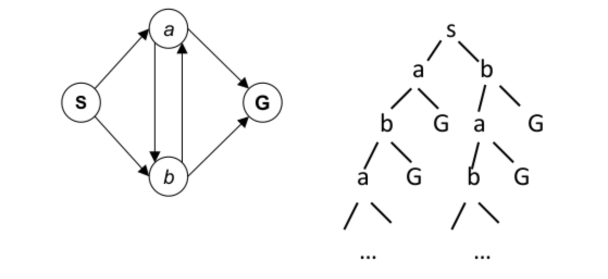
Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another

  $\Longrightarrow$ same state explored more than once

- Failure to detect repeated states can:
  - cause infinite loops
  - turn linear problem into exponential



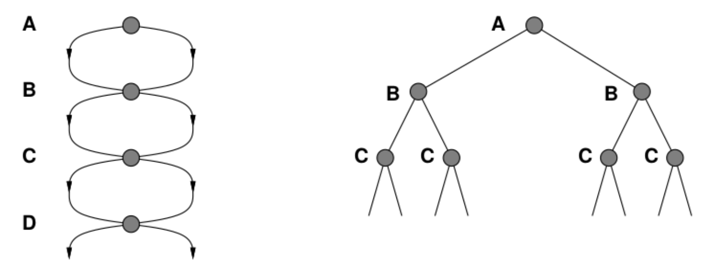
(© S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another

  $\Longrightarrow$ same state explored more than once
- Failure to detect repeated states can:
  - cause infinite loops
  - turn linear problem into exponential



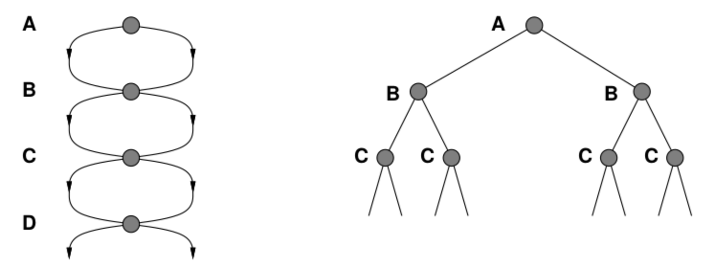
(© S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Repeated states & Redundant Paths

- redundant paths occur when there is more than one way to get from one state to another

  $\Longrightarrow$ same state explored more than once
- Failure to detect repeated states can:
  - cause infinite loops
  - turn linear problem into exponential



(© S. Russell & P. Norwig, AIMA)

Moral: Algorithms that forget their history are doomed to repeat it!

Graph Search Algorithms

Graph Search: Basic idea

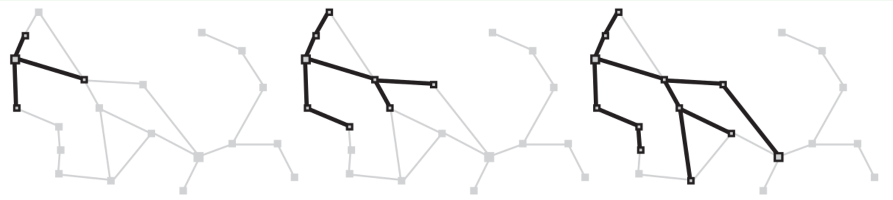
- add a data structure which remembers every expanded node
  - a.k.a. explored set or closed list
  - typically a hash table (access $O(1)$)
- do not expand a node if it already occurs in explored set

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Graph Search Algorithms: Example

Graph search on the Romania trip problem

- (at each stage each path extended by one step)
- two states become dead-end

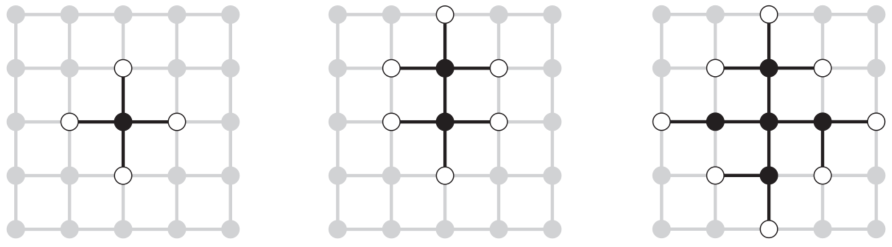


(© S. Russell & P. Norwig, AIMA)

Graph Search Algorithms: Example

Separation Property of graph search: the frontier separates the state-space graph into the explored region and the unexplored region

Graph search on a rectangular-grid problem



(© S. Russell & P. Norwig, AIMA)

Implementation: States vs. Nodes

- A state is a representation of a physical configuration
- A node is a data structure constituting part of a search tree
  - includes fields: state, parent, action, path cost $g(x)$

$\Longrightarrow$ node $\neq$ state

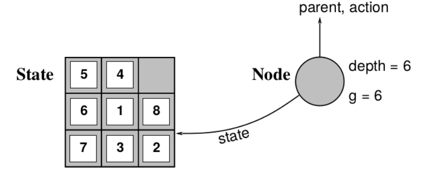
- It is easy to compute a child node from its parent

Implementation: States vs. Nodes

- A state is a representation of a physical configuration
- A node is a data structure constituting part of a search tree
  - includes fields: state, parent, action, path cost $g(x)$

$\Longrightarrow$ node $\neq$ state

- It is easy to compute a child node from its parent



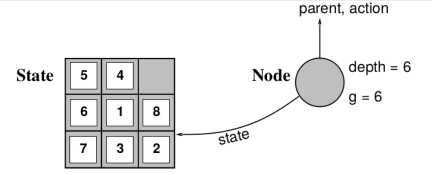
(© S. Russell & P. Norwig, AIMA)

Implementation: States vs. Nodes

- A state is a representation of a physical configuration
- A node is a data structure constituting part of a search tree
  - includes fields: state, parent, action, path cost $g(x)$

$\Longrightarrow$ node $\neq$ state

- It is easy to compute a child node from its parent



function CHILD-NODE(*problem*, *parent*, *action*) **returns** a node
  return a node with
    STATE = *problem*.RESULT(*parent*.STATE, *action*),
    PARENT = *parent*, ACTION = *action*,
    PATH-COST = *parent*.PATH-COST + *problem*.STEP-COST(*parent*.STATE, *action*)

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
  - First-in-First-Out, FIFO (aka “queue”): $O(1)$ access
  - Last-in-First-Out, LIFO (aka “stack”): $O(1)$ access
  - Best-First-out (aka “priority queue”): $O(log(n))$ access
- Two primitives:
  - IsEmpty(queue): returns true iff there are no more elements
  - Pop(queue): removes and returns the first element of the queue
  - Insert(element,queue): inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
  - IsThere(element,hash): returns true iff element is in the hash
  - Insert(element,hash): inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
  - First-in-First-Out, FIFO (aka "queue"): $O(1)$ access
  - Last-in-First-Out, LIFO (aka "stack"): $O(1)$ access
  - Best-First-out (aka "priority queue"): $O(log(n))$ access
- Two primitives:
  - IsEmpty(queue): returns true iff there are no more elements
  - Pop(queue): removes and returns the first element of the queue
  - Insert(element,queue): inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
  - IsThere(element,hash): returns true iff element is in the hash
  - Insert(element,hash): inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
  - First-in-First-Out, FIFO (aka "queue"): $O(1)$ access
  - Last-in-First-Out, LIFO (aka "stack"): $O(1)$ access
  - Best-First-out (aka "priority queue"): $O(log(n))$ access
- Two primitives:
  - ISEMPTY(QUEUE): returns true iff there are no more elements
  - POP(QUEUE): removes and returns the first element of the queue
  - INSERT(ELEMENT,QUEUE): inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
  - ISTHERE(ELEMENT,HASH): returns true iff element is in the hash
  - INSERT(ELEMENT,HASH): inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
  - First-in-First-Out, FIFO (aka "queue"): $O(1)$ access
  - Last-in-First-Out, LIFO (aka "stack"): $O(1)$ access
  - Best-First-out (aka "priority queue"): $O(log(n))$ access
- Two primitives:
  - ISEMPTY(QUEUE): returns true iff there are no more elements
  - POP(QUEUE): removes and returns the first element of the queue
  - INSERT(ELEMENT,QUEUE): inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
  - ISTHERE(ELEMENT,HASH): returns true iff element is in the hash
  - INSERT(ELEMENT,HASH): inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
  - First-in-First-Out, FIFO (aka "queue"): $O(1)$ access
  - Last-in-First-Out, LIFO (aka "stack"): $O(1)$ access
  - Best-First-out (aka "priority queue"): $O(log(n))$ access
- Two primitives:
  - ISEMPTY(QUEUE): returns true iff there are no more elements
  - POP(QUEUE): removes and returns the first element of the queue
  - INSERT(ELEMENT,QUEUE): inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
  - ISTHERE(ELEMENT,HASH): returns true iff element is in the hash
  - INSERT(ELEMENT,HASH): inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
  - First-in-First-Out, FIFO (aka "queue"): $O(1)$ access
  - Last-in-First-Out, LIFO (aka "stack"): $O(1)$ access
  - Best-First-out (aka "priority queue"): $O(log(n))$ access
- Two primitives:
  - ISEMPTY(QUEUE): returns true iff there are no more elements
  - POP(QUEUE): removes and returns the first element of the queue
  - INSERT(ELEMENT,QUEUE): inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
  - ISTHERE(ELEMENT,HASH): returns true iff element is in the hash
  - INSERT(ELEMENT,HASH): inserts element into hash
- Choice of hash function critical for efficiency

Implementation: Frontier and Explored

Frontier/Fringe

- Implemented as a Queue:
  - First-in-First-Out, FIFO (aka "queue"): $O(1)$ access
  - Last-in-First-Out, LIFO (aka "stack"): $O(1)$ access
  - Best-First-out (aka "priority queue"): $O(log(n))$ access
- Two primitives:
  - ISEMPTY(QUEUE): returns true iff there are no more elements
  - POP(QUEUE): removes and returns the first element of the queue
  - INSERT(ELEMENT,QUEUE): inserts an element into queue

Explored

- Implemented as a Hash Table: $O(1)$ access
- Three primitives:
  - ISTHERE(ELEMENT,HASH): returns true iff element is in the hash
  - INSERT(ELEMENT,HASH): inserts element into hash
- Choice of hash function critical for efficiency

Uninformed vs. Informed Search Strategies

Strategies: Two possibilities

- Uninformed strategies (a.k.a. blind strategies)
  - do not use any domain knowledge
  - apply rules arbitrarily and do an exhaustive search strategy

  $\Longrightarrow$ impractical for some complex problems.
- Informed strategies
  - use domain knowledge
  - apply rules following heuristics (driven by domain knowledge)

  $\Longrightarrow$ practical for many complex problems.

Uninformed vs. Informed Search Strategies

Strategies: Two possibilities

- Uninformed strategies (a.k.a. blind strategies)
  - do not use any domain knowledge
  - apply rules arbitrarily and do an exhaustive search strategy

  $\Longrightarrow$ impractical for some complex problems.
- Informed strategies
  - use domain knowledge
  - apply rules following heuristics (driven by domain knowledge)

  $\Longrightarrow$ practical for many complex problems.

Uninformed vs. Informed Search Strategies

Strategies: Two possibilities

- Uninformed strategies (a.k.a. blind strategies)
  - do not use any domain knowledge
  - apply rules arbitrarily and do an exhaustive search strategy

  $\Longrightarrow$ impractical for some complex problems.
- Informed strategies
  - use domain knowledge
  - apply rules following heuristics (driven by domain knowledge)

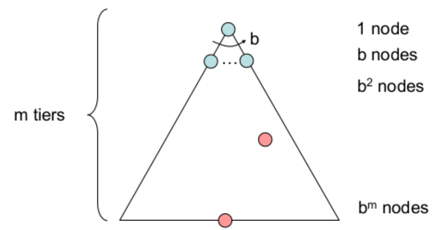
  $\Longrightarrow$ practical for many complex problems.

Outline

Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

$\Longrightarrow$ # nodes: $1 + b + b^2 + \ldots + b^m = O(b^m)$

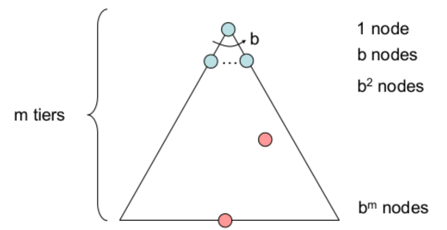


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

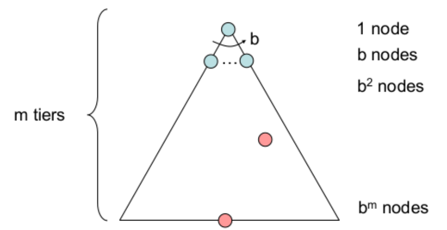
$\Longrightarrow$ # nodes: $1 + b + b^2 + \ldots + b^m = O(b^m)$



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

$\Longrightarrow$ # nodes: $1 + b + b^2 + \ldots + b^m = O(b^m)$

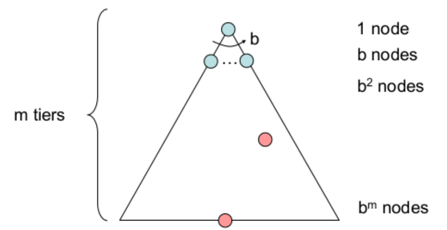


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

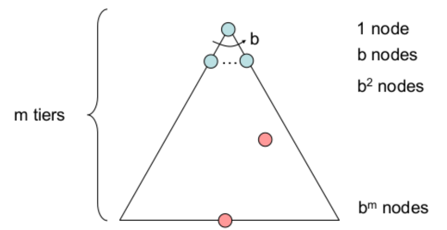
$\Longrightarrow$ # nodes: $1 + b + b^2 + \ldots + b^m = O(b^m)$



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

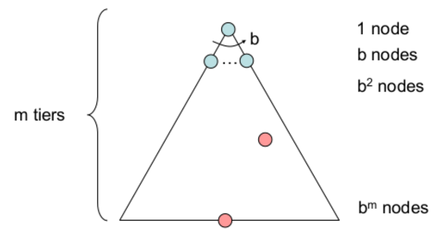
$\Longrightarrow$ # nodes: $1 + b + b^2 + \ldots + b^m = O(b^m)$



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

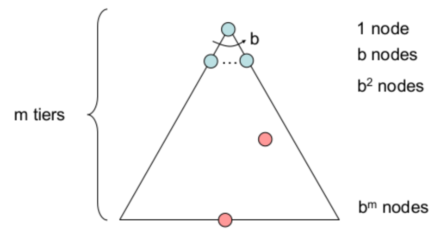
$\Longrightarrow$ # nodes: $1 + b + b^2 + ... + b^m = O(b^m)$



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

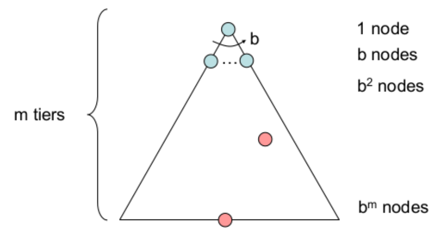
$\Longrightarrow$ # nodes: $1 + b + b^2 + \ldots + b^m = O(b^m)$



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

$\Longrightarrow$ # nodes: $1 + b + b^2 + ... + b^m = O(b^m)$

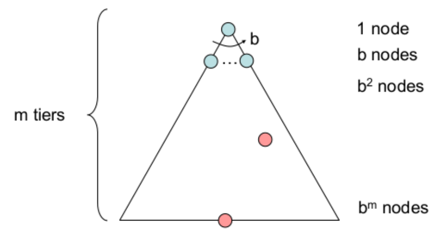


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

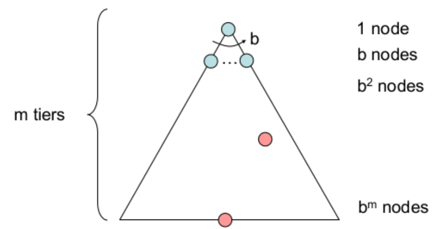
$\Longrightarrow$ # nodes: $1 + b + b^2 + ... + b^m = O(b^m)$



Evaluating Search Strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: how many steps to find a solution?
  - space complexity: how much memory is needed?
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be $+\infty$)

$\Longrightarrow$ # nodes: $1 + b + b^2 + ... + b^m = O(b^m)$



Uninformed Search Strategies

Uninformed strategies

Use only the information available in the problem definition

- Different uninformed search stategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search & Iterative-deepening search
- Defined by the access strategy of the frontier/fringe (i.e. the order of node expansion)

Outline

Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue

$\Longrightarrow$ novel successors pushed to the end of the queue

Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue

  $\Longrightarrow$ novel successors pushed to the end of the queue

Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue

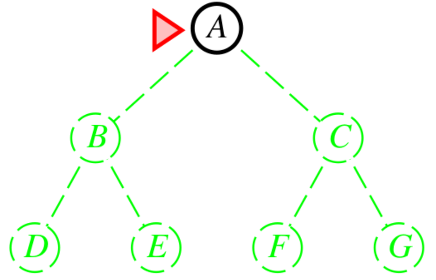
$\Longrightarrow$ novel successors pushed to the end of the queue

Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue

$\Longrightarrow$ novel successors pushed to the end of the queue

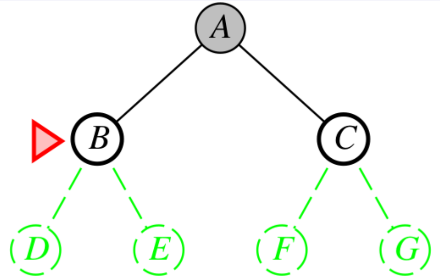


Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue

  $\Longrightarrow$ novel successors pushed to the end of the queue

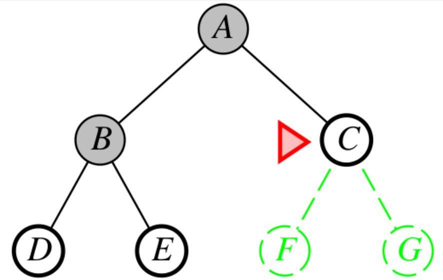


Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue

  $\Longrightarrow$ novel successors pushed to the end of the queue

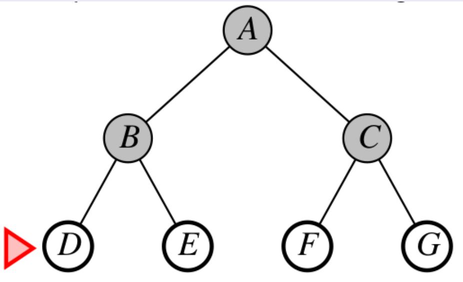


Breadth-First Search Strategy (BFS)

Breadth-First Search

- Idea: Expand first the shallowest unexpanded nodes
- Implementation: frontier/fringe implemented as a FIFO queue

$\Longrightarrow$ novel successors pushed to the end of the queue



Breadth-First Search Strategy (BFS) [cont.]

BFS, Graph version (Tree version without "explored")

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

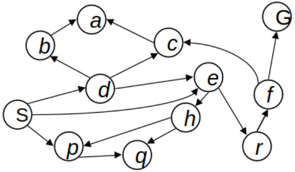

Note: the goal test is applied to each node when it is generated, rather than when it is selected for expansion
$\Longrightarrow$ solution detected 1 layer earlier

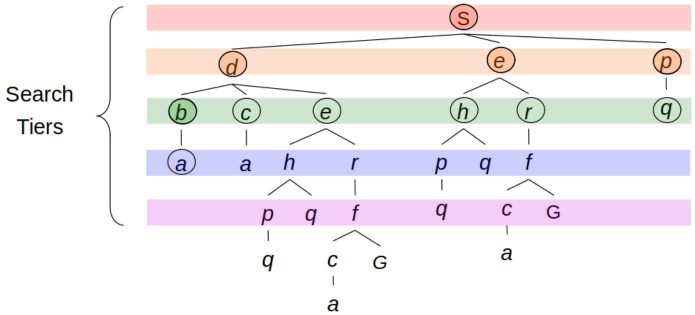
Breadth-First Search: Tiers

State space is explored by tiers (tree version)

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

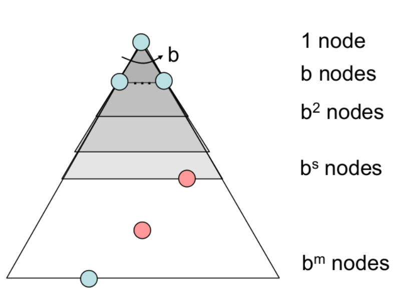




Breadth-First Search (BFS): Properties

d: depth of shallowest solution

- How many steps?
  - processes all nodes above shallowest solution
    $\Longrightarrow$ takes $O(b^d)$ time
- How much memory?
  - max frontier size: b^d nodes
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - if solution exists, b^d finite
    $\Longrightarrow$ Yes
- Is it optimal?
  - if and only if all costs are 1
    $\Longrightarrow$ shallowest solution



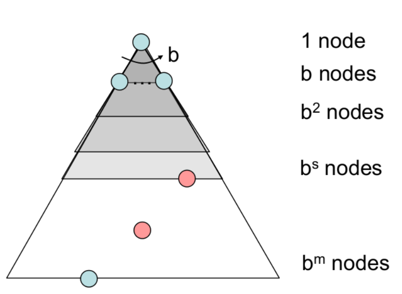
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem for breadth-first search

Breadth-First Search (BFS): Properties

d: depth of shallowest solution

- How many steps?
  - processes all nodes above shallowest solution
    $\Longrightarrow$ takes $O(b^d)$ time
- How much memory?
  - max frontier size: b^d nodes
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - if solution exists, b^d finite
    $\Longrightarrow$ Yes
- Is it optimal?
  - if and only if all costs are 1
    $\Longrightarrow$ shallowest solution



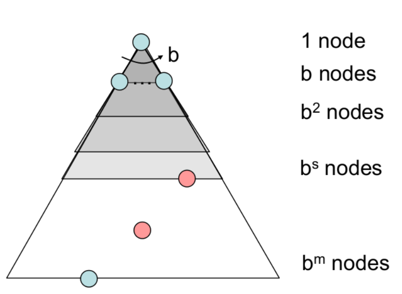
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem for breadth-first search

Breadth-First Search (BFS): Properties

d: depth of shallowest solution

- How many steps?
  - processes all nodes above shallowest solution
    $\Longrightarrow$ takes $O(b^d)$ time
- How much memory?
  - max frontier size: b^d nodes
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - if solution exists, b^d finite
    $\Longrightarrow$ Yes
- Is it optimal?
  - if and only if all costs are 1
    $\Longrightarrow$ shallowest solution



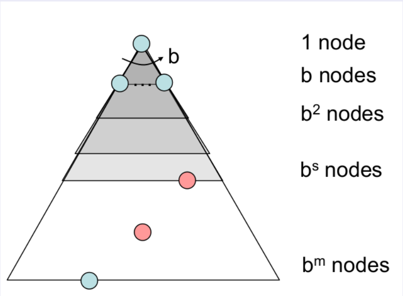
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem for breadth-first search

Breadth-First Search (BFS): Properties

d: depth of shallowest solution

- How many steps?
  - processes all nodes above shallowest solution
    $\Longrightarrow$ takes $O(b^d)$ time
- How much memory?
  - max frontier size: b^d nodes
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - if solution exists, b^d finite
    $\Longrightarrow$ Yes
- Is it optimal?
  - if and only if all costs are 1
    $\Longrightarrow$ shallowest solution



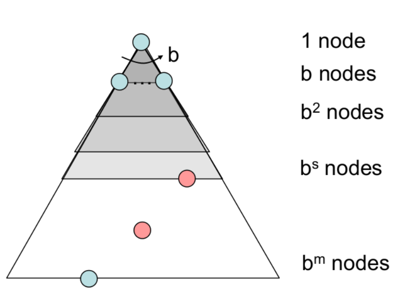
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem for breadth-first search

Breadth-First Search (BFS): Properties

d: depth of shallowest solution

- How many steps?
  - processes all nodes above shallowest solution
    $\Longrightarrow$ takes $O(b^d)$ time
- How much memory?
  - max frontier size: b^d nodes
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - if solution exists, b^d finite
    $\Longrightarrow$ Yes
- Is it optimal?
  - if and only if all costs are 1
    $\Longrightarrow$ shallowest solution



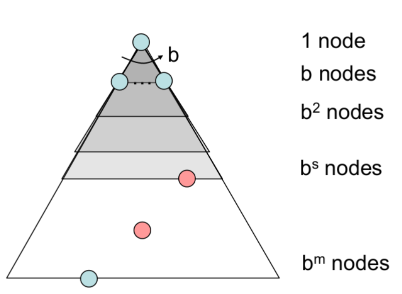
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem for breadth-first search

Breadth-First Search (BFS): Properties

d: depth of shallowest solution

- How many steps?
  - processes all nodes above shallowest solution
    $\Longrightarrow$ takes $O(b^d)$ time
- How much memory?
  - max frontier size: b^d nodes
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - if solution exists, b^d finite
    $\Longrightarrow$ Yes
- Is it optimal?
  - if and only if all costs are 1
    $\Longrightarrow$ shallowest solution



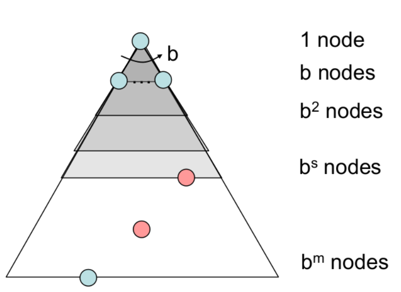
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem for breadth-first search

Breadth-First Search (BFS): Properties

d: depth of shallowest solution

- How many steps?
  - processes all nodes above shallowest solution
    $\Longrightarrow$ takes $O(b^d)$ time
- How much memory?
  - max frontier size: b^d nodes
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - if solution exists, b^d finite
    $\Longrightarrow$ Yes
- Is it optimal?
  - if and only if all costs are 1
    $\Longrightarrow$ shallowest solution



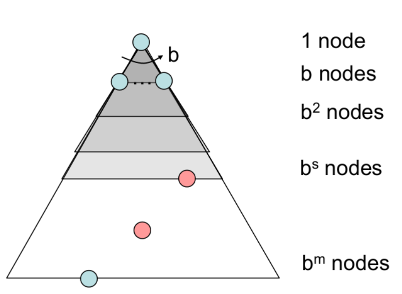
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem for breadth-first search

Breadth-First Search (BFS): Properties

d: depth of shallowest solution

- How many steps?
  - processes all nodes above shallowest solution
    $\Longrightarrow$ takes $O(b^d)$ time
- How much memory?
  - max frontier size: b^d nodes
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - if solution exists, b^d finite
    $\Longrightarrow$ Yes
- Is it optimal?
  - if and only if all costs are 1
    $\Longrightarrow$ shallowest solution



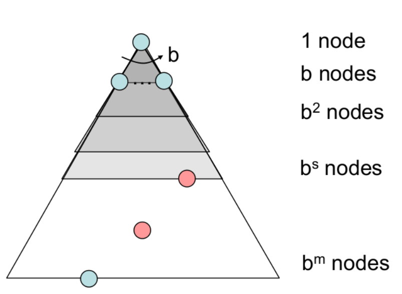
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem for breadth-first search

Breadth-First Search (BFS): Properties

d: depth of shallowest solution

- How many steps?
  - processes all nodes above shallowest solution
    $\Longrightarrow$ takes $O(b^d)$ time
- How much memory?
  - max frontier size: b^d nodes
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - if solution exists, b^d finite
    $\Longrightarrow$ Yes
- Is it optimal?
  - if and only if all costs are 1
    $\Longrightarrow$ shallowest solution



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem for breadth-first search

Breadth-First Search (BFS): Time and Memory

- Assume:
  - 1 million nodes generated per second
  - 1 node requires 1000 bytes of storage
  - branching factor $b = 10$

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Memory requirements bigger problem for BFS than execution time

Outline

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$

  $\Longrightarrow$ novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

(Courtesy of Michela Milano, UniBO)

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$

  $\Longrightarrow$ novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

(Courtesy of Michela Milano, UniBO)

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$

  $\Longrightarrow$ novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

(Courtesy of Michela Milano, UniBO)

Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$

  $\Longrightarrow$ novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal

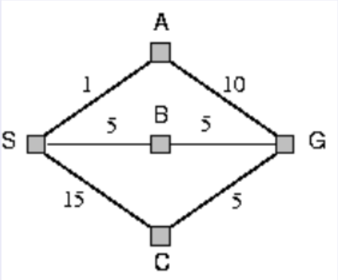
(Courtesy of Michela Milano, UniBO)

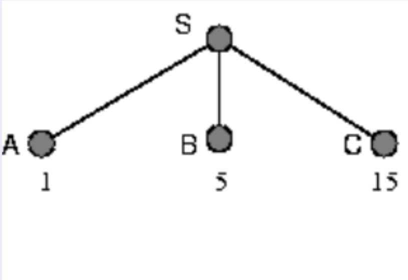
Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$

  $\Longrightarrow$ novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal





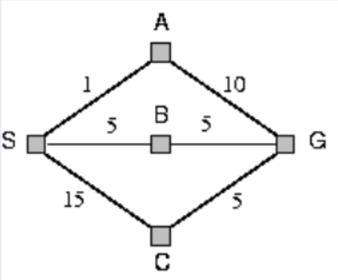
(Courtesy of Michela Milano, UniBO)

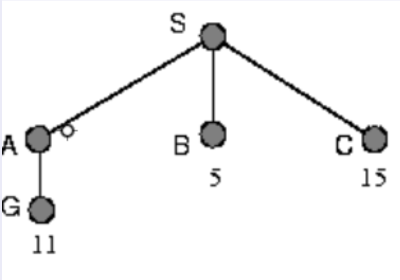
Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$

  $\Longrightarrow$ novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal





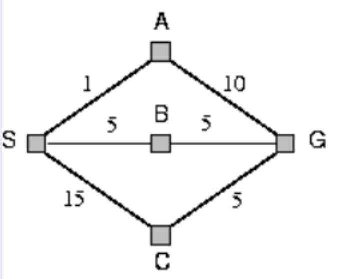
(Courtesy of Michela Milano, UniBO)

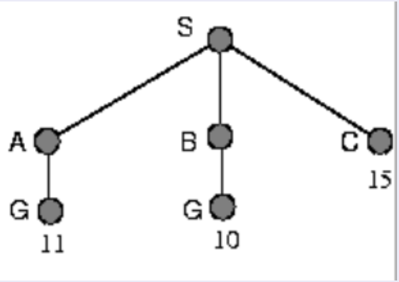
Uniform-Cost Search Strategy (UCS)

Uniform-Cost Search

- Idea: Expand first the node with lowest path cost $g(n)$
- Implementation: frontier/fringe implemented as a priority queue ordered by $g()$

  $\Longrightarrow$ novel nearest successors pushed to the top of the queue
- similar to BFS if step costs are all equal





(Courtesy of Michela Milano, UniBO)

Uniform-Cost Search Strategy (UCS) [cont.]

UCS, Graph version (Tree version without “explored”)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

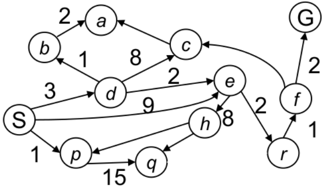

- apply the goal test to a node when it is selected for expansion rather than when it is first generated
- replace in the frontier a node with same state but worse path cost

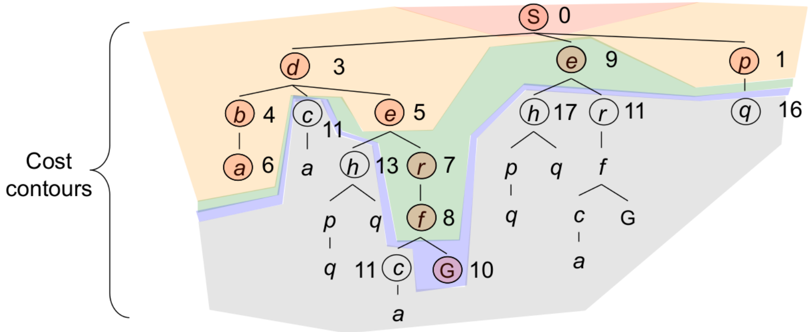
$\Longrightarrow$ avoid generating suboptimal paths

Uniform-Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)



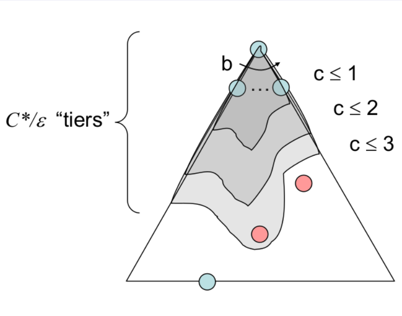


Uniform-Cost Search (UCS): Properties

C^*: cost of cheapest solution; ϵ: minimum arc cost

$\Longrightarrow 1 + \lfloor C^*/\epsilon \rfloor$ "effective depth"

- How many steps?
  - processes all nodes costing less than cheapest solution $\Longrightarrow$ takes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time
- How much memory?
  - max frontier size: $b^{1+\lfloor C^*/\epsilon \rfloor}$ $\Longrightarrow O(b^{1+\lfloor C^*/\epsilon \rfloor})$ memory size
- Is it complete?
  - if solution exists, finite cost $\Longrightarrow$ Yes
- Is it optimal?
  - Yes



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

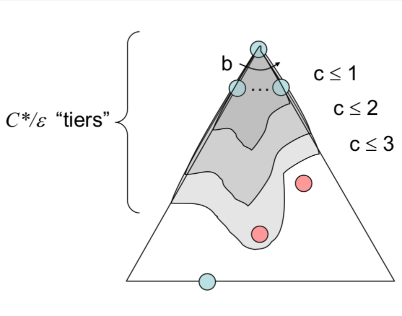
Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^*: cost of cheapest solution; ϵ: minimum arc cost

$\Longrightarrow 1 + \lfloor C^*/\epsilon \rfloor$ "effective depth"

- How many steps?
  - processes all nodes costing less than cheapest solution $\Longrightarrow$ takes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time
- How much memory?
  - max frontier size: $b^{1+\lfloor C^*/\epsilon \rfloor}$ $\Longrightarrow O(b^{1+\lfloor C^*/\epsilon \rfloor})$ memory size
- Is it complete?
  - if solution exists, finite cost $\Longrightarrow$ Yes
- Is it optimal?
  - Yes



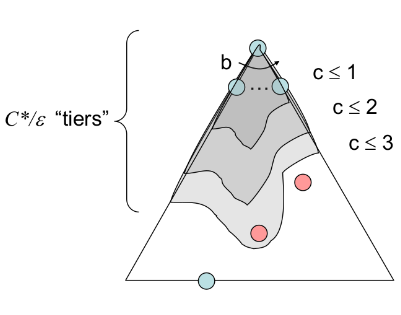
Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^*: cost of cheapest solution; ϵ: minimum arc cost

$\Longrightarrow 1 + \lfloor C^*/\epsilon \rfloor$ "effective depth"

- How many steps?
  - processes all nodes costing less than cheapest solution $\Longrightarrow$ takes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time
- How much memory?
  - max frontier size: $b^{1+\lfloor C^*/\epsilon \rfloor}$ $\Longrightarrow$ $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ memory size
- Is it complete?
  - if solution exists, finite cost $\Longrightarrow$ Yes
- Is it optimal?
  - Yes



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

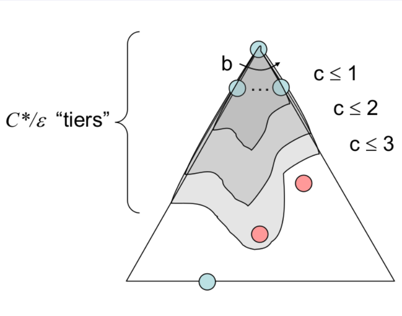
Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^*: cost of cheapest solution; ϵ: minimum arc cost

$\Longrightarrow 1 + \lfloor C^*/\epsilon \rfloor$ "effective depth"

- How many steps?
  - processes all nodes costing less than cheapest solution $\Longrightarrow$ takes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time
- How much memory?
  - max frontier size: $b^{1+\lfloor C^*/\epsilon \rfloor}$ $\Longrightarrow$ $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ memory size
- Is it complete?
  - if solution exists, finite cost $\Longrightarrow$ Yes
- Is it optimal?
  - Yes



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

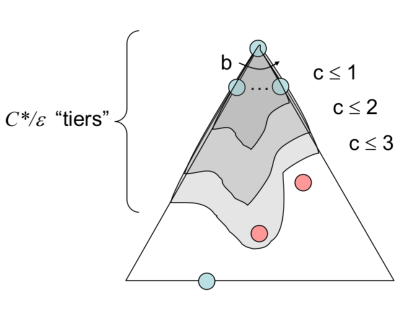
Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^*: cost of cheapest solution; ϵ: minimum arc cost

$\Longrightarrow 1 + \lfloor C^*/\epsilon \rfloor$ "effective depth"

- How many steps?
  - processes all nodes costing less than cheapest solution $\Longrightarrow$ takes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time
- How much memory?
  - max frontier size: $b^{1+\lfloor C^*/\epsilon \rfloor}$ $\Longrightarrow$ $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ memory size
- Is it complete?
  - if solution exists, finite cost $\Longrightarrow$ Yes
- Is it optimal?
  - Yes



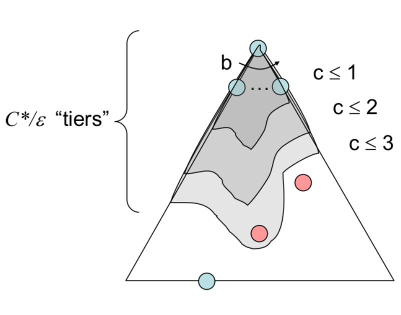
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^*: cost of cheapest solution; ϵ: minimum arc cost
$\Longrightarrow 1 + \lfloor C^*/\epsilon \rfloor$ "effective depth"

- How many steps?
  - processes all nodes costing less than cheapest solution $\Longrightarrow$ takes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time
- How much memory?
  - max frontier size: $b^{1+\lfloor C^*/\epsilon \rfloor}$ $\Longrightarrow$ $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ memory size
- Is it complete?
  - if solution exists, finite cost $\Longrightarrow$ Yes
- Is it optimal?
  - Yes

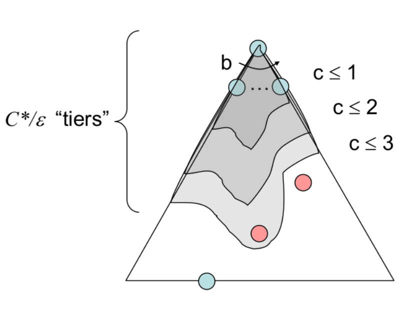


Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^*: cost of cheapest solution; ϵ: minimum arc cost
$\Longrightarrow 1 + \lfloor C^*/\epsilon \rfloor$ "effective depth"

- How many steps?
  - processes all nodes costing less than cheapest solution $\Longrightarrow$ takes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time
- How much memory?
  - max frontier size: $b^{1+\lfloor C^*/\epsilon \rfloor}$ $\Longrightarrow$ $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ memory size
- Is it complete?
  - if solution exists, finite cost $\Longrightarrow$ Yes
- Is it optimal?
  - Yes



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

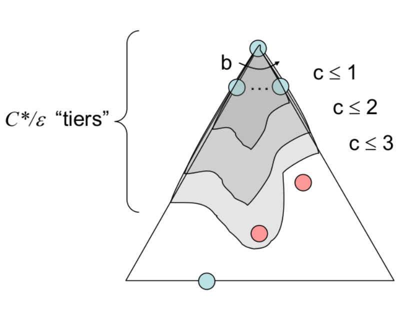
Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^*: cost of cheapest solution; ϵ: minimum arc cost

$\Longrightarrow 1 + \lfloor C^*/\epsilon \rfloor$ "effective depth"

- How many steps?
  - processes all nodes costing less than cheapest solution $\Longrightarrow$ takes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time
- How much memory?
  - max frontier size: $b^{1+\lfloor C^*/\epsilon \rfloor}$ $\Longrightarrow$ $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ memory size
- Is it complete?
  - if solution exists, finite cost $\Longrightarrow$ Yes
- Is it optimal?
  - Yes



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

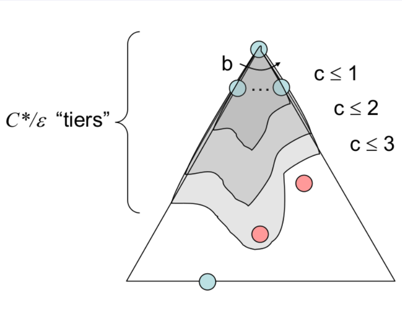
Memory requirement is a major problem also for uniform-cost search

Uniform-Cost Search (UCS): Properties

C^*: cost of cheapest solution; ϵ: minimum arc cost

$\Longrightarrow 1 + \lfloor C^*/\epsilon \rfloor$ "effective depth"

- How many steps?
  - processes all nodes costing less than cheapest solution $\Longrightarrow$ takes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time
- How much memory?
  - max frontier size: $b^{1+\lfloor C^*/\epsilon \rfloor}$ $\Longrightarrow$ $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ memory size
- Is it complete?
  - if solution exists, finite cost $\Longrightarrow$ Yes
- Is it optimal?
  - Yes



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for uniform-cost search

Outline

Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

$\Longrightarrow$ novel successors pushed to the top of the stack

Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack

(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

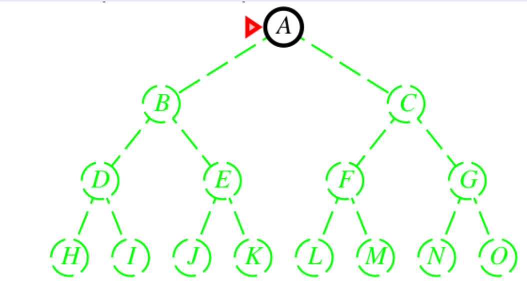
  $\Longrightarrow$ novel successors pushed to the top of the stack

Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack

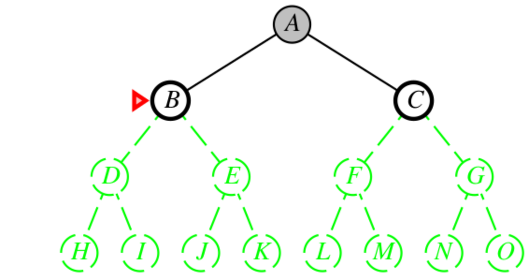


Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack

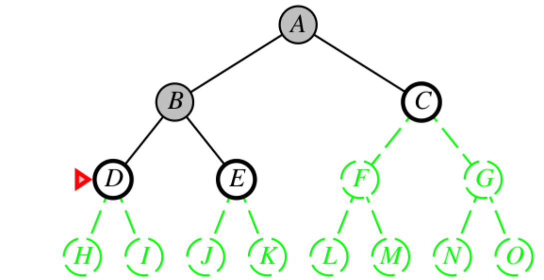


Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack

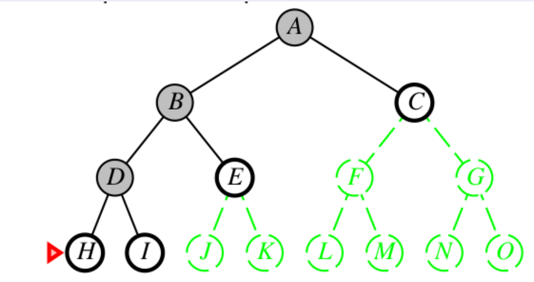


Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack



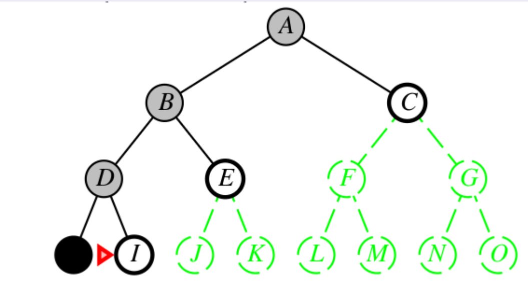
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack



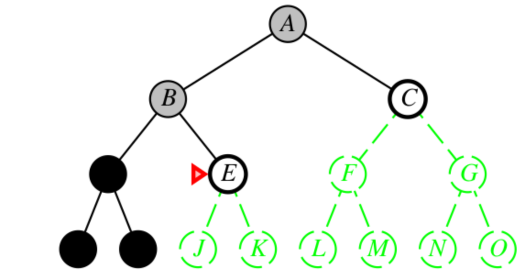
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack



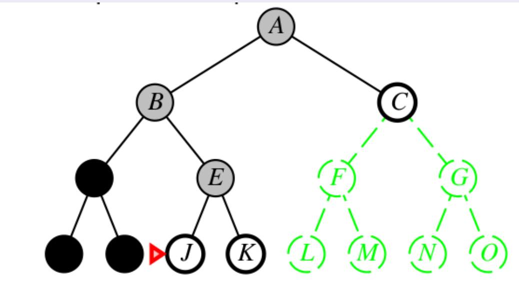
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack

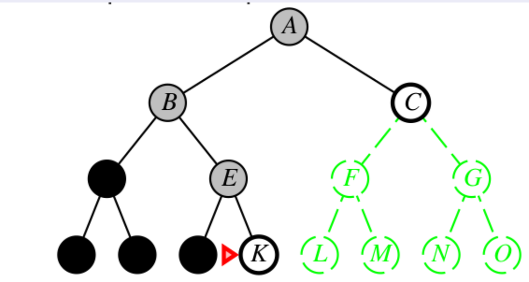


Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack

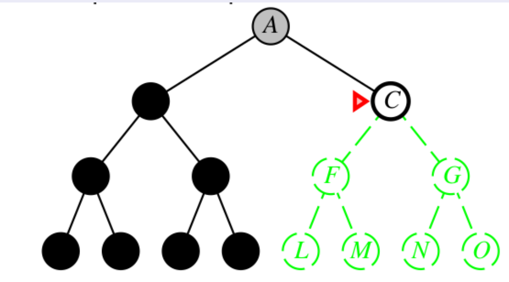


Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack

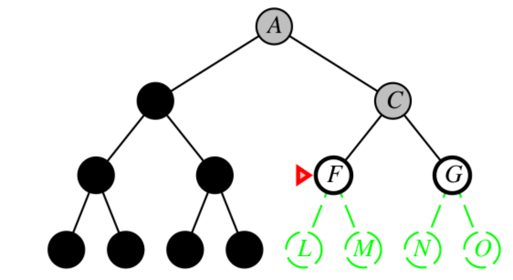


Depth-First Search Strategy (DFS)

Depth-First Search

- Idea: Expand first the deepest unexpanded nodes
- Implementation: frontier/fringe implemented as a LIFO queue (aka stack)

  $\Longrightarrow$ novel successors pushed to the top of the stack



Depth-First Search

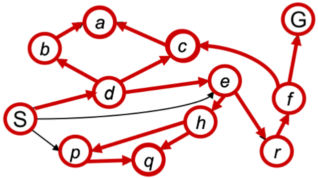
DFS on a Graph

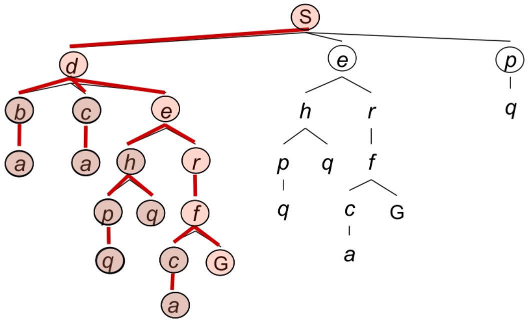
Similar to BFS, using a LIFO access for frontier/fringe rather than FIFO.

Depth-First Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack

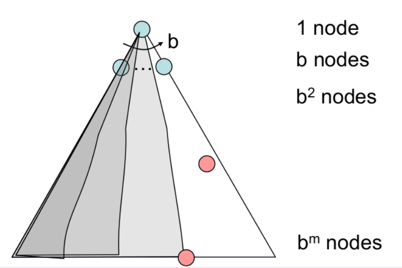




(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! ⟹ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root ⟹ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost ⟹ "leftmost" solution

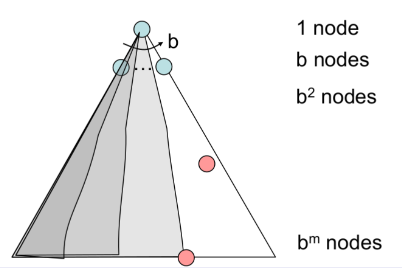


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
⟹ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution

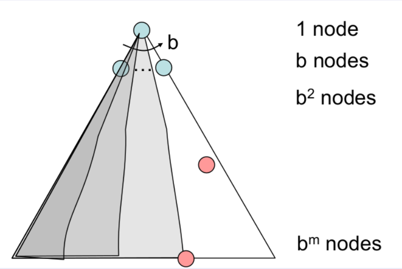


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution

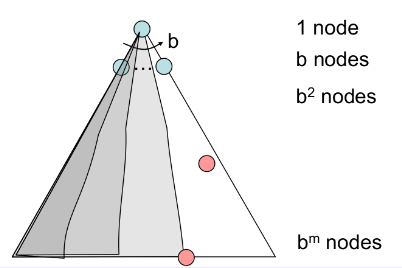


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution

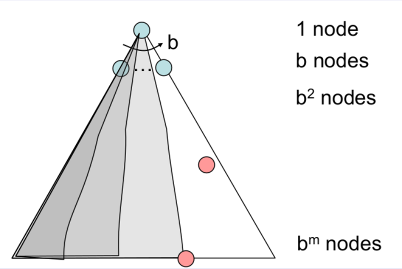


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

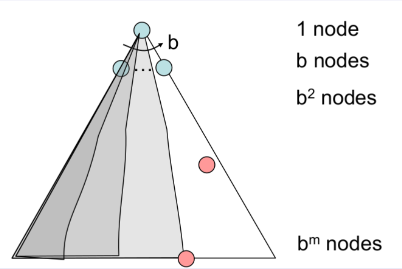
- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution



Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution

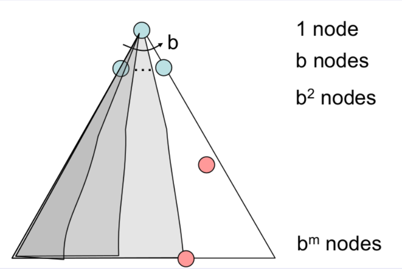


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution

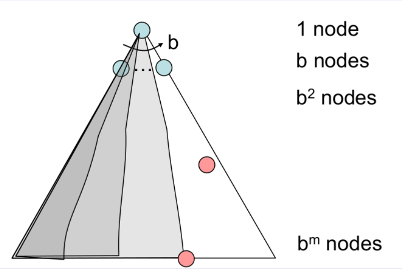


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution

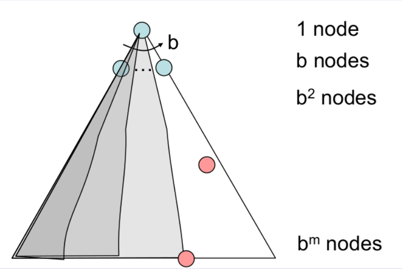


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution

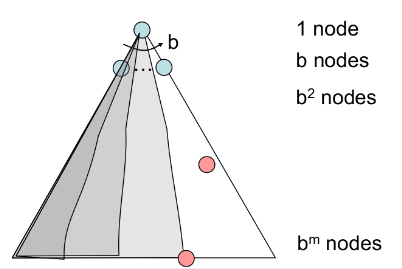


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution

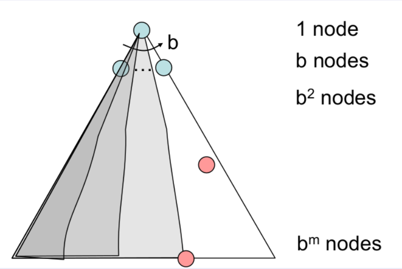


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ "leftmost" solution

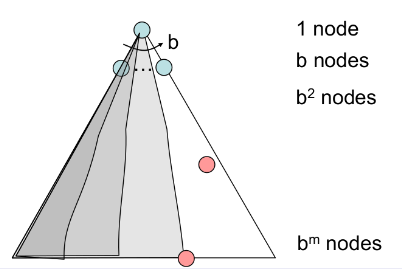


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

Depth-First Search (DFS): Properties

- How many steps?
  - could process the whole tree! $\Longrightarrow$ if m finite, takes $O(b^m)$ time
- How much memory?
  - only siblings on path to root $\Longrightarrow$ $O(bm)$ memory size
- Is it complete?
  - if infinite state space: no
  - if finite state space:
    - graph version: yes
    - tree version: only if we prevent loops
- Is it optimal?
  - No, regardless of depth/cost $\Longrightarrow$ “leftmost” solution



Memory requirement much better than BFS: $O(bm)$ vs. $O(b^d)$!
$\Longrightarrow$ typically preferred to BFS

A Variant of DFS: Backtracking Search

Backtracking Search

- Idea: only one successor is generated at the time
  - each partially-expanded node remembers which successor to generate next
  - generate a successor by modifying the current state description, rather than copying it firs
  - Applied in CSP, SAT/SMT and Logic Programming

$\Longrightarrow$ only $O(m)$ memory is needed rather than $O(bm)$

A Variant of DFS: Backtracking Search

Backtracking Search

- Idea: only one successor is generated at the time
  - each partially-expanded node remembers which successor to generate next
  - generate a successor by modifying the current state description, rather than copying it firs
  - Applied in CSP, SAT/SMT and Logic Programming

$\Longrightarrow$ only $O(m)$ memory is needed rather than $O(bm)$

Outline

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
  - i.e., nodes at depth l treated as having no successors
  - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
  $\Longrightarrow$ allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
  - e.g., if maximum node distance in a graph (diameter) is known
  - Ex: Romania trip: 9 steps
- Drawbacks (d: depth of the shallowest goal):
  - if $d > l \Longrightarrow$ incomplete
  - if $d < l \Longrightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
  - i.e., nodes at depth l treated as having no successors
  - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
  $\Longrightarrow$ allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
  - e.g., if maximum node distance in a graph (diameter) is known
  - Ex: Romania trip: 9 steps
- Drawbacks (d: depth of the shallowest goal):
  - if $d > l \Longrightarrow$ incomplete
  - if $d < l \Longrightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
  - i.e., nodes at depth l treated as having no successors
  - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
  $\Longrightarrow$ allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
  - e.g., if maximum node distance in a graph (diameter) is known
  - Ex: Romania trip: 9 steps
- Drawbacks (d: depth of the shallowest goal):
  - if $d > l \Longrightarrow$ incomplete
  - if $d < l \Longrightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
  - i.e., nodes at depth l treated as having no successors
  - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
  $\Longrightarrow$ allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
  - e.g., if maximum node distance in a graph (diameter) is known
  - Ex: Romania trip: 9 steps
- Drawbacks (d: depth of the shallowest goal):
  - if $d > l \Longrightarrow$ incomplete
  - if $d < l \Longrightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
  - i.e., nodes at depth l treated as having no successors
  - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
  $\Longrightarrow$ allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
  - e.g., if maximum node distance in a graph (diameter) is known
  - Ex: Romania trip: 9 steps
- Drawbacks (d: depth of the shallowest goal):
  - if $d > l \Longrightarrow$ incomplete
  - if $d < l \Longrightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
  - i.e., nodes at depth l treated as having no successors
  - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
  $\Longrightarrow$ allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
  - e.g., if maximum node distance in a graph (diameter) is known
  - Ex: Romania trip: 9 steps
- Drawbacks (d: depth of the shallowest goal):
  - if $d > l \Longrightarrow$ incomplete
  - if $d < l \Longrightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy

Depth-Limited Search (DLS)

- Idea: depth-first search with depth limit l
  - i.e., nodes at depth l treated as having no successors
  - DFS is DLS with $l = +\infty$
- solves the infinite-path problem of DFS
  $\Longrightarrow$ allows DFS deal with infinite-state spaces
- useful also if maximum-depth is known by domain knowledge
  - e.g., if maximum node distance in a graph (diameter) is known
  - Ex: Romania trip: 9 steps
- Drawbacks (d: depth of the shallowest goal):
  - if $d > l \Longrightarrow$ incomplete
  - if $d < l \Longrightarrow$ takes $O(b^l)$ instead of $O(b^d)$ steps

Depth-Limited Search (DLS) Strategy [cont.]

Recursive DLS

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   else if limit = 0 then return cutoff
   else
       cutoff_occurred? ← false
       for each action in problem.ACTIONS(node.STATE) do
           child ← CHILD-NODE(problem, node, action)
           result ← RECURSIVE-DLS(child, problem, limit − 1)
           if result = cutoff then cutoff_occurred? ← true
           else if result ≠ failure then return result
       if cutoff_occurred? then return cutoff else return failure
```

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3...$
- combines the advantages of breadth- and depth-first strategies
  - complete (like BFS)
  - takes $O(b^d)$ steps (like BFS and DFS)
  - requires $O(bd)$ memory (like DFS)
  - explores a single branch at a time (like DFS)
  - optimal only if step cost = 1
    - optimal variants exist: iterative-lengthening search (see AIMA)
- The favorite search strategy when the search space is very large and depth is not known

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3...$
- combines the advantages of breadth- and depth-first strategies
  - complete (like BFS)
  - takes $O(b^d)$ steps (like BFS and DFS)
  - requires $O(bd)$ memory (like DFS)
  - explores a single branch at a time (like DFS)
  - optimal only if step cost = 1
    - optimal variants exist: iterative-lengthening search (see AIMA)
- The favorite search strategy when the search space is very large and depth is not known

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3...$
- combines the advantages of breadth- and depth-first strategies
  - complete (like BFS)
  - takes $O(b^d)$ steps (like BFS and DFS)
  - requires $O(bd)$ memory (like DFS)
  - explores a single branch at a time (like DFS)
  - optimal only if step cost = 1
    - optimal variants exist: iterative-lengthening search (see AIMA)
- The favorite search strategy when the search space is very large and depth is not known

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3...$
- combines the advantages of breadth- and depth-first strategies
  - complete (like BFS)
  - takes $O(b^d)$ steps (like BFS and DFS)
  - requires $O(bd)$ memory (like DFS)
  - explores a single branch at a time (like DFS)
  - optimal only if step cost = 1
    - optimal variants exist: iterative-lengthening search (see AIMA)
- The favorite search strategy when the search space is very large and depth is not known

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3...$
- combines the advantages of breadth- and depth-first strategies
  - complete (like BFS)
  - takes $O(b^d)$ steps (like BFS and DFS)
  - requires $O(bd)$ memory (like DFS)
  - explores a single branch at a time (like DFS)
  - optimal only if step cost = 1
    - optimal variants exist: iterative-lengthening search (see AIMA)
- The favorite search strategy when the search space is very large and depth is not known

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3...$
- combines the advantages of breadth- and depth-first strategies
  - complete (like BFS)
  - takes $O(b^d)$ steps (like BFS and DFS)
  - requires $O(bd)$ memory (like DFS)
  - explores a single branch at a time (like DFS)
  - optimal only if step cost = 1
    - optimal variants exist: iterative-lengthening search (see AIMA)
- The favorite search strategy when the search space is very large and depth is not known

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3...$
- combines the advantages of breadth- and depth-first strategies
  - complete (like BFS)
  - takes $O(b^d)$ steps (like BFS and DFS)
  - requires $O(bd)$ memory (like DFS)
  - explores a single branch at a time (like DFS)
  - optimal only if step cost = 1
    - optimal variants exist: iterative-lengthening search (see AIMA)
- The favorite search strategy when the search space is very large and depth is not known

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Iterative-Deepening Search Strategy (IDS)

Iterative-Deepening Search

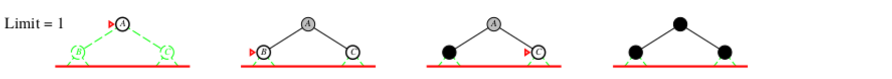
- Idea: call iteratively DLS for increasing depths $l = 0, 1, 2, 3...$
- combines the advantages of breadth- and depth-first strategies
  - complete (like BFS)
  - takes $O(b^d)$ steps (like BFS and DFS)
  - requires $O(bd)$ memory (like DFS)
  - explores a single branch at a time (like DFS)
  - optimal only if step cost = 1
    - optimal variants exist: iterative-lengthening search (see AIMA)
- The favorite search strategy when the search space is very large and depth is not known

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

Iterative-Deepening Search: Example

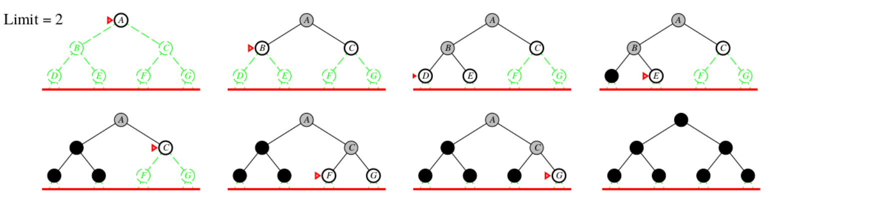
(© S. Russell & P. Norwig, AIMA)

Iterative-Deepening Search: Example



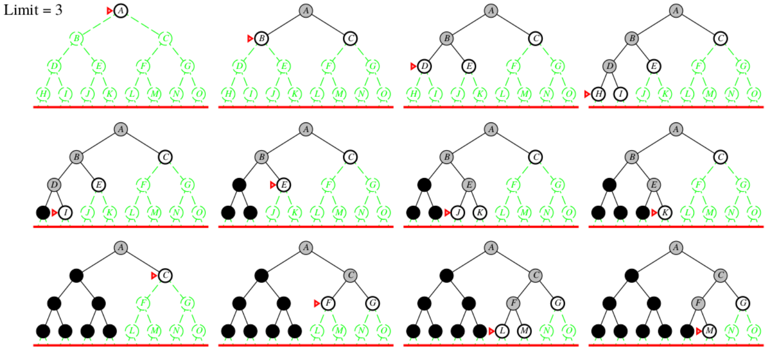
(© S. Russell & P. Norwig, AIMA)

Iterative-Deepening Search: Example



(© S. Russell & P. Norwig, AIMA)

Iterative-Deepening Search: Example



(© S. Russell & P. Norwig, AIMA)

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why "only" $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
  - depth 1 (b nodes): repeated d times
  - depth 2 (b^2 nodes): repeated $d-1$ times
  - ...
  - depth d (b^d nodes): repeated 1 time

$\Longrightarrow$ The total number of generated nodes is:

$$N(IDS) = (d)b^1 + (d-1)b^2 + ... + (1)b^d = O(b^d)$$
$$N(BFS) = b^1 + b^2 + ... + b^d = O(b^d)$$

- Ex: with $b = 10$ and $d = 5$:

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000$$
$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

$\Longrightarrow$ not significantly worse than BFS

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why "only" $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
  - depth 1 (b nodes): repeated d times
  - depth 2 (b^2 nodes): repeated $d-1$ times
  - ...
  - depth d (b^d nodes): repeated 1 time

$\Longrightarrow$ The total number of generated nodes is:

$$N(IDS) = (d)b^1 + (d-1)b^2 + ... + (1)b^d = O(b^d)$$
$$N(BFS) = b^1 + b^2 + ... + b^d = O(b^d)$$

- Ex: with $b = 10$ and $d = 5$:

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000$$
$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

$\Longrightarrow$ not significantly worse than BFS

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why "only" $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
  - depth 1 (b nodes): repeated d times
  - depth 2 (b^2 nodes): repeated $d-1$ times
  - ...
  - depth d (b^d nodes): repeated 1 time

$\Longrightarrow$ The total number of generated nodes is:

$$N(IDS) = (d)b^1 + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$$
$$N(BFS) = b^1 + b^2 + \ldots + b^d = O(b^d)$$

- Ex: with $b = 10$ and $d = 5$:

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000$$
$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

$\Longrightarrow$ not significantly worse than BFS

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why "only" $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
  - depth 1 (b nodes): repeated d times
  - depth 2 (b^2 nodes): repeated $d-1$ times
  - ...
  - depth d (b^d nodes): repeated 1 time

$\Longrightarrow$ The total number of generated nodes is:

$$N(IDS) = (d)b^1 + (d-1)b^2 + ... + (1)b^d = O(b^d)$$
$$N(BFS) = b^1 + b^2 + ... + b^d = O(b^d)$$

- Ex: with $b = 10$ and $d = 5$:

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000$$
$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

$\Longrightarrow$ not significantly worse than BFS

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why "only" $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
  - depth 1 (b nodes): repeated d times
  - depth 2 (b^2 nodes): repeated $d-1$ times
  - ...
  - depth d (b^d nodes): repeated 1 time

$\Longrightarrow$ The total number of generated nodes is:

$$N(IDS) = (d)b^1 + (d-1)b^2 + ... + (1)b^d = O(b^d)$$
$$N(BFS) = b^1 + b^2 + ... + b^d = O(b^d)$$

- Ex: with $b = 10$ and $d = 5$:

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000$$
$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

$\Longrightarrow$ not significantly worse than BFS

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why "only" $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
  - depth 1 (b nodes): repeated d times
  - depth 2 (b^2 nodes): repeated $d-1$ times
  - ...
  - depth d (b^d nodes): repeated 1 time

$\Longrightarrow$ The total number of generated nodes is:

$$N(IDS) = (d)b^1 + (d-1)b^2 + ... + (1)b^d = O(b^d)$$
$$N(BFS) = b^1 + b^2 + ... + b^d = O(b^d)$$

- Ex: with $b = 10$ and $d = 5$:

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000$$
$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

$\Longrightarrow$ not significantly worse than BFS

Iterative-Deepening Search Strategy (IDS) [cont.]

Remark: Why "only" $O(b^d)$ steps?

- may seem wasteful since states are generated multiple times...
- ... however, only a small fraction of nodes are multiply generated
- number of repeatedly-generated nodes decreases exponentially with number of repetitions
  - depth 1 (b nodes): repeated d times
  - depth 2 (b^2 nodes): repeated $d-1$ times
  - ...
  - depth d (b^d nodes): repeated 1 time

$\Longrightarrow$ The total number of generated nodes is:

$$N(IDS) = (d)b^1 + (d-1)b^2 + ... + (1)b^d = O(b^d)$$
$$N(BFS) = b^1 + b^2 + ... + b^d = O(b^d)$$

- Ex: with $b = 10$ and $d = 5$:

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,000$$
$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

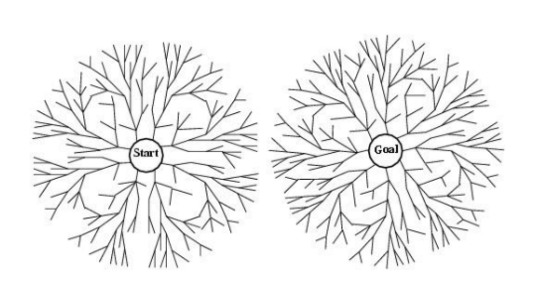
$\Longrightarrow$ not significantly worse than BFS

Bidirectional Search [hints]

- Idea: Two simultaneous searches:
  - forward: from start node
  - backward: from goal node

  checking if the node belongs to the other frontier before expansion
- Rationale: $b^{d/2} + b^{d/2} << b^d$

  $\Longrightarrow$ number of steps and memory consumption are $\approx 2b^{d/2}$
- backward search can be tricky in some cases (e.g. 8-queens)



Uninformed Search Strategies: Comparison

Evaluation of tree-search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

[a]: complete if b is finite
[b]: complete if step costs $\geq \epsilon$ for some positive ϵ
[c]: optimal if step costs are all identical
[d]: if both directions use breadth-first search

For graph searches, the main differences are:

- depth-first search is complete for finite-state spaces
- space & time complexities are bounded by the state space size

Outline

Informed Search Strategies

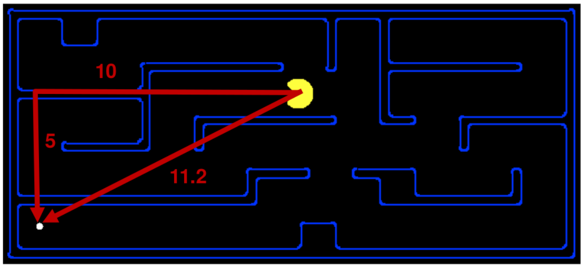
Some general principles

- The intelligence of a system cannot be measured only in terms of search capacity, but in the ability to use knowledge about the problem to reduce/mitigate the combinatorial explosion
- If the system has some control on the order in which candidate solutions are generated, then it is useful to use this order so that actual solutions have a high chance to appear earlier
- Intelligence, for a system with limited processing capacity, is the wise choice of what to do next

Heuristic search and heuristic functions

Heuristic search and heuristic functions

- Idea: don't ignore the goal when selecting nodes
- Intuition: often there is extra knowledge that can be used to guide the search towards the goal: heuristics
- A heuristic is
  - a function $h(n)$ that estimates how close a state n is to a goal
  - designed for a particular search problem
  - Ex Manhattan distance, Euclidean distance for pathing



Best-first Search Strategies

General approach of informed search: Best-first search

- Best-first search: node selected for expansion based on an evaluation function $f(n)$
  - represent a cost estimate $\Longrightarrow$ choose node which appears best
  - implemented like uniform-cost search, with f instead of g

  $\Longrightarrow$ the frontier is a priority queue sorted in decreasing order of $h(n)$
  - both tree-based and graph-based versions
  - most often f includes a heuristic function h(n)
- Heuristic function $h(n) \in \mathbb{R}^+$: estimated cost of the cheapest path from the state at node n to a goal state
  - $h(n) \geq 0 \; \forall n$
  - If G is goal, then $h(G) = 0$
  - implements extra domain knowledge
  - depends only on state, not on node (e.g., independent on paths)
- Main strategies:
  - Greedy search
  - A^* search

Best-first Search Strategies

General approach of informed search: Best-first search

- Best-first search: node selected for expansion based on an evaluation function $f(n)$
  - represent a cost estimate $\Longrightarrow$ choose node which appears best
  - implemented like uniform-cost search, with f instead of g

  $\Longrightarrow$ the frontier is a priority queue sorted in decreasing order of $h(n)$
  - both tree-based and graph-based versions
  - most often f includes a heuristic function h(n)
- Heuristic function $h(n) \in \mathbb{R}^+$: estimated cost of the cheapest path from the state at node n to a goal state
  - $h(n) \geq 0 \ \forall n$
  - If G is goal, then $h(G) = 0$
  - implements extra domain knowledge
  - depends only on state, not on node (e.g., independent on paths)
- Main strategies:
  - Greedy search
  - A^* search

Best-first Search Strategies

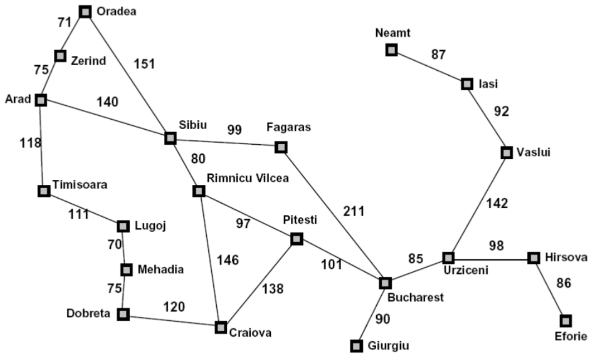
General approach of informed search: Best-first search

- Best-first search: node selected for expansion based on an evaluation function $f(n)$
  - represent a cost estimate $\Longrightarrow$ choose node which appears best
  - implemented like uniform-cost search, with f instead of g

  $\Longrightarrow$ the frontier is a priority queue sorted in decreasing order of $h(n)$
  - both tree-based and graph-based versions
  - most often f includes a heuristic function h(n)
- Heuristic function $h(n) \in \mathbb{R}^+$: estimated cost of the cheapest path from the state at node n to a goal state
  - $h(n) \geq 0 \ \forall n$
  - If G is goal, then $h(G) = 0$
  - implements extra domain knowledge
  - depends only on state, not on node (e.g., independent on paths)
- Main strategies:
  - Greedy search
  - A^* search

Example: Straight-Line Distance $h_{SLD}(n)$

- $h(n) \stackrel{\text{def}}{=} h_{SLD}(n)$: straight-line distance heuristic
  - different from actual minimum-path dinstance
  - cannot be computed from the problem description itself



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Outline

Greedy Best-First Search Strategy (GBFS)

Greedy Best-First Search (aka Greedy Search)

- Idea: Expand first the node n with lowest estimate cost to the closest goal, $h(n)$
- Implementation: same as uniform-cost search, with $g(n) \stackrel{\text{def}}{=} h(n)$
  $\Longrightarrow$ expands the node that appears to be closest to goal
- both tree and graph versions

Greedy Best-First Search Strategy (GBFS)

Greedy Best-First Search (aka Greedy Search)

- Idea: Expand first the node n with lowest estimate cost to the closest goal, $h(n)$
- Implementation: same as uniform-cost search, with $g(n) \stackrel{\text{def}}{=} h(n)$
  $\Longrightarrow$ expands the node that appears to be closest to goal
- both tree and graph versions

Greedy Best-First Search Strategy (GBFS)

Greedy Best-First Search (aka Greedy Search)

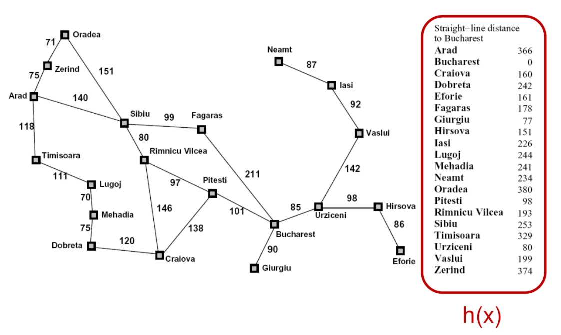
- Idea: Expand first the node n with lowest estimate cost to the closest goal, $h(n)$
- Implementation: same as uniform-cost search, with $g(n) \stackrel{\text{def}}{=} h(n)$
  $\Longrightarrow$ expands the node that appears to be closest to goal
- both tree and graph versions

Greedy Best-First Search Strategy (GBFS)

Greedy Best-First Search (aka Greedy Search)

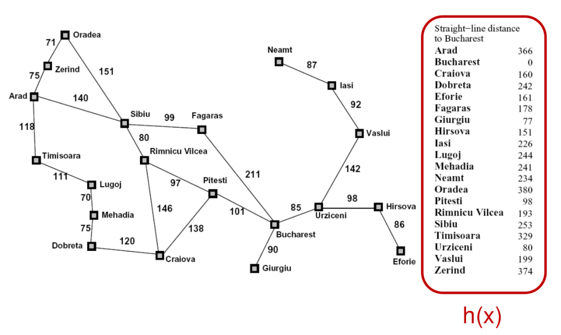
- Idea: Expand first the node n with lowest estimate cost to the closest goal, $h(n)$
- Implementation: same as uniform-cost search, with $g(n) \stackrel{\text{def}}{=} h(n)$
  $\Longrightarrow$ expands the node that appears to be closest to goal
- both tree and graph versions

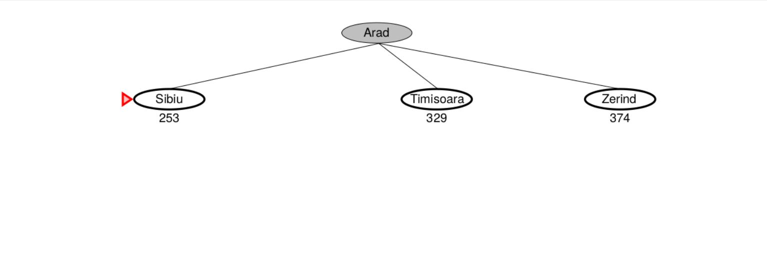
Greedy Best-First Search Strategy: Example



(© S. Russell & P. Norwig, AIMA)

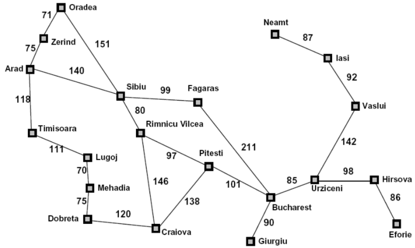
Greedy Best-First Search Strategy: Example





(© S. Russell & P. Norwig, AIMA)

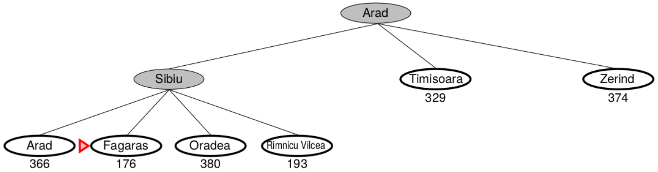
Greedy Best-First Search Strategy: Example



Straight−line distance to Bucharest

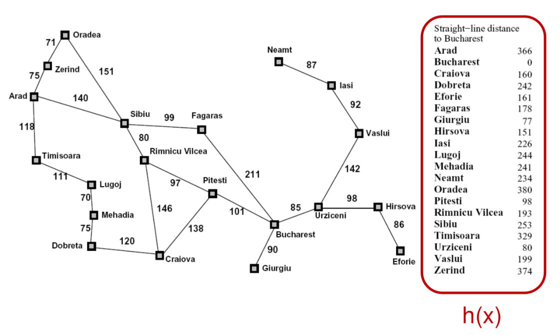
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

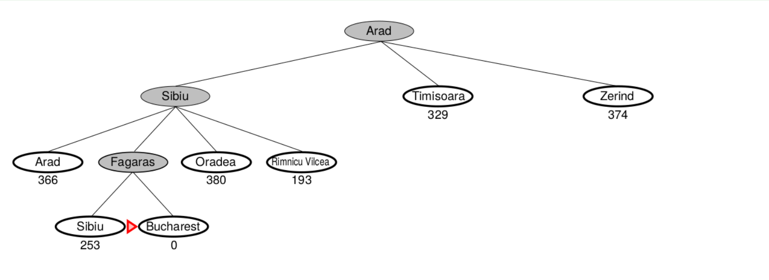
h(x)



(© S. Russell & P. Norwig, AIMA)

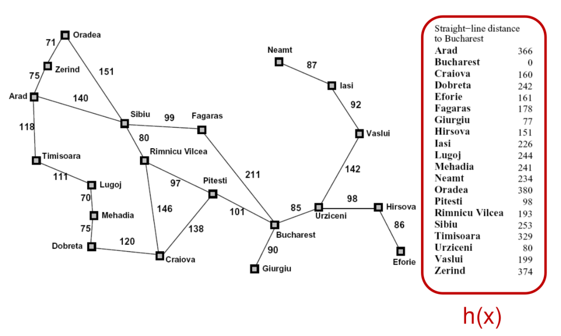
Greedy Best-First Search Strategy: Example

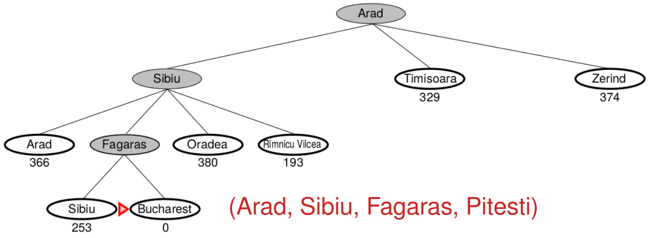




(© S. Russell & P. Norwig, AIMA)

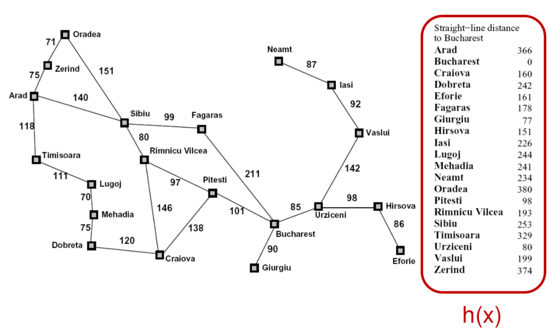
Greedy Best-First Search Strategy: Example

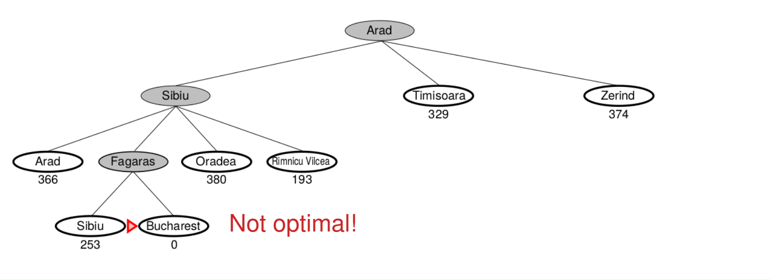




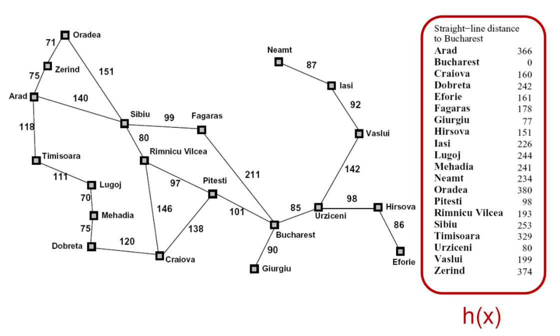
(Arad, Sibiu, Fagaras, Pitesti)

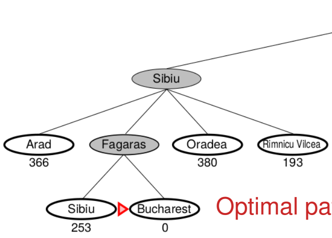
Greedy Best-First Search Strategy: Example





Greedy Best-First Search Strategy: Example



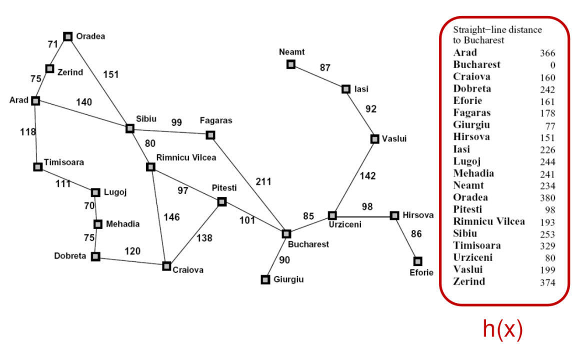


Optimal path: (Arad, Sibiu, Rimnicu Vilcea, Pitesti)

(© S. Russell & P. Norvig, AIMA)

Greedy Best-First Search: (Non-)Optimality

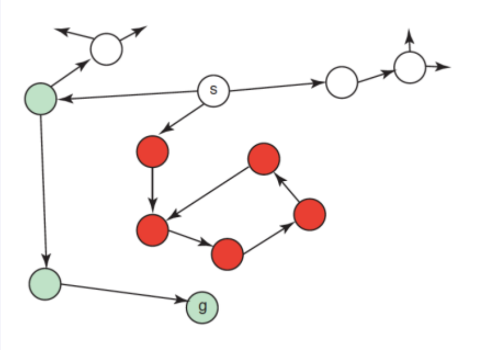
- Greedy best-first search is not optimal
  - it is not guaranteed to find the best solution
  - it is not guaranteed to find the best path toward a solution
- picks the node with minimum (estimated) distance to goal, regardless the cost to reach it
  - Ex: when in Sibiu, it picks Fagaras rather than Rimnicu Vicea



(© S. Russell & P. Norwig, AIMA)

Greedy Best-First Search: (In-)Completeness

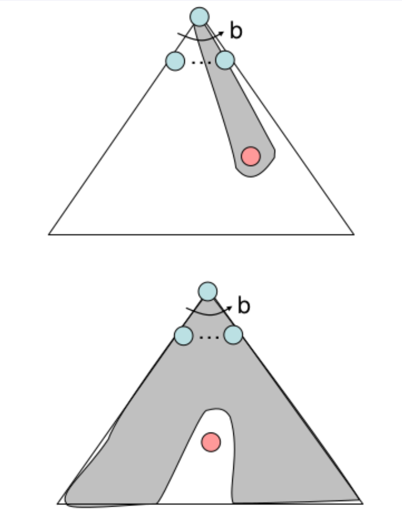
- Tree-based Greedy best-first search is not complete
  - may lead to infinite loops
- Graph-based version complete (if state space finite)
- substantially same completeness issues as DFS



(Courtesy of Maria Simi, UniPI)

Greedy Best-First Search (GBFS): Properties

- How many steps?
  - in worst cases may explore all states
    $\Longrightarrow$ takes $O(b^d)$ time
    if good heuristics:
    $\Longrightarrow$ may give good improvements
- How much memory?
  - max frontier size: b^d
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - tree: no
    graph: yes if space finite
- Is it optimal?
  - No

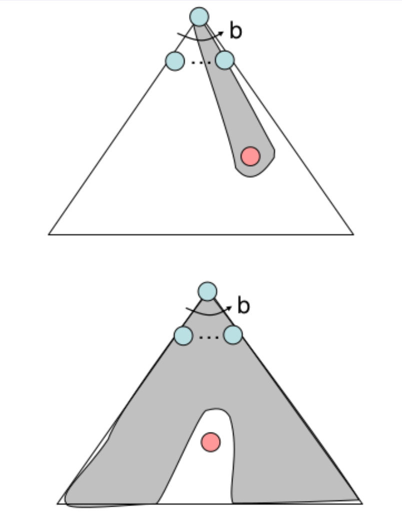


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for GBFS

Greedy Best-First Search (GBFS): Properties

- How many steps?
  - in worst cases may explore all states
    $\Longrightarrow$ takes $O(b^d)$ time
    if good heuristics:
    $\Longrightarrow$ may give good improvements
- How much memory?
  - max frontier size: b^d
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - tree: no
    graph: yes if space finite
- Is it optimal?
  - No

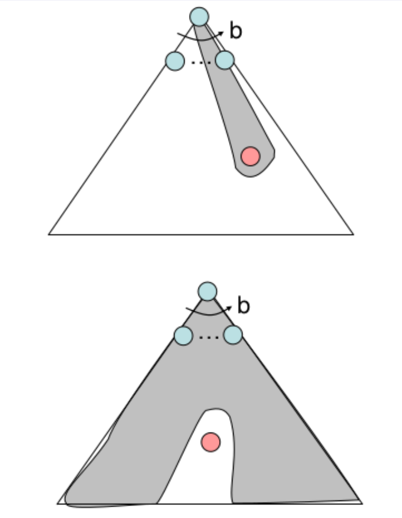


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for GBFS

Greedy Best-First Search (GBFS): Properties

- How many steps?
  - in worst cases may explore all states
    $\Longrightarrow$ takes $O(b^d)$ time
    if good heuristics:
    $\Longrightarrow$ may give good improvements
- How much memory?
  - max frontier size: b^d
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - tree: no
    graph: yes if space finite
- Is it optimal?
  - No

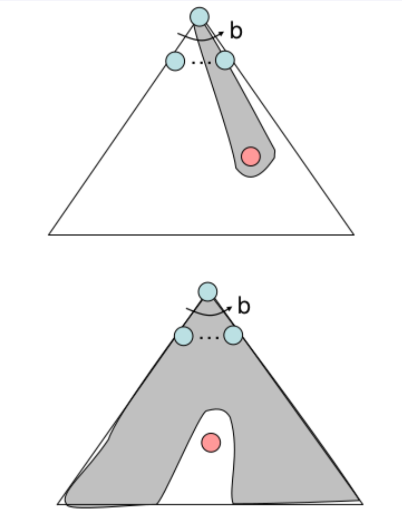


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for GBFS

Greedy Best-First Search (GBFS): Properties

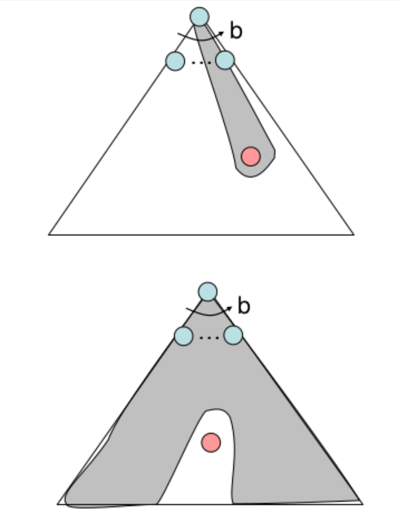
- How many steps?
  - in worst cases may explore all states
    $\Longrightarrow$ takes $O(b^d)$ time
    if good heuristics:
    $\Longrightarrow$ may give good improvements
- How much memory?
  - max frontier size: b^d
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - tree: no
    graph: yes if space finite
- Is it optimal?
  - No



Memory requirement is a major problem also for GBFS

Greedy Best-First Search (GBFS): Properties

- How many steps?
  - in worst cases may explore all states
    $\Longrightarrow$ takes $O(b^d)$ time
    if good heuristics:
    $\Longrightarrow$ may give good improvements
- How much memory?
  - max frontier size: b^d
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - tree: no
    graph: yes if space finite
- Is it optimal?
  - No

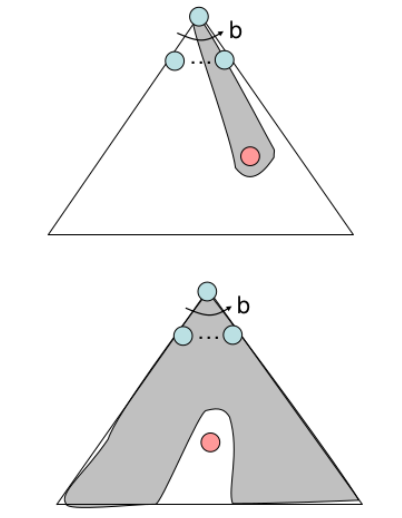


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for GBFS

Greedy Best-First Search (GBFS): Properties

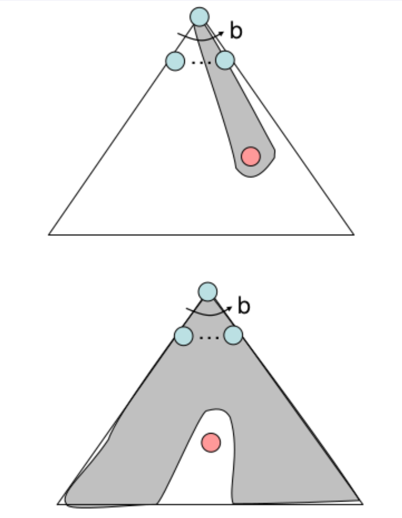
- How many steps?
  - in worst cases may explore all states
    $\Longrightarrow$ takes $O(b^d)$ time
    if good heuristics:
    $\Longrightarrow$ may give good improvements
- How much memory?
  - max frontier size: b^d
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - tree: no
    graph: yes if space finite
- Is it optimal?
  - No



Memory requirement is a major problem also for GBFS

Greedy Best-First Search (GBFS): Properties

- How many steps?
  - in worst cases may explore all states
    $\Longrightarrow$ takes $O(b^d)$ time
    if good heuristics:
    $\Longrightarrow$ may give good improvements
- How much memory?
  - max frontier size: b^d
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - tree: no
    graph: yes if space finite
- Is it optimal?
  - No

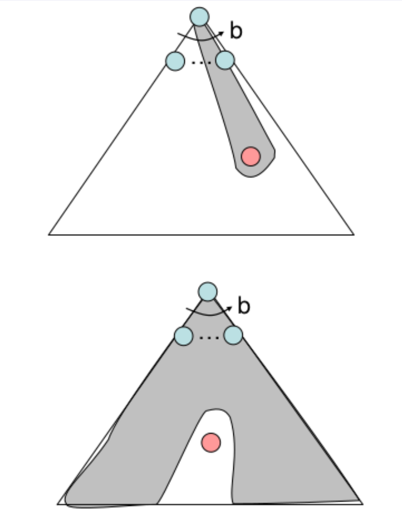


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for GBFS

Greedy Best-First Search (GBFS): Properties

- How many steps?
  - in worst cases may explore all states
    $\Longrightarrow$ takes $O(b^d)$ time
    if good heuristics:
    $\Longrightarrow$ may give good improvements
- How much memory?
  - max frontier size: b^d
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - tree: no
    graph: yes if space finite
- Is it optimal?
  - No

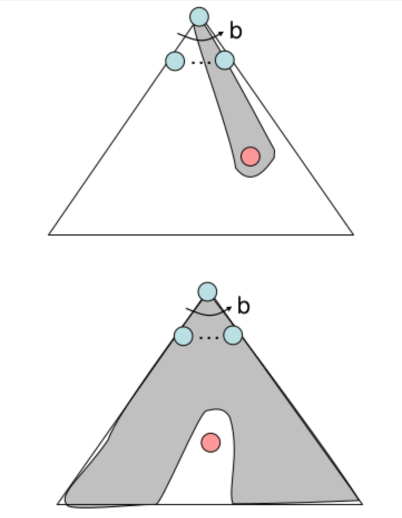


(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for GBFS

Greedy Best-First Search (GBFS): Properties

- How many steps?
  - in worst cases may explore all states
    $\Longrightarrow$ takes $O(b^d)$ time
    if good heuristics:
    $\Longrightarrow$ may give good improvements
- How much memory?
  - max frontier size: b^d
    $\Longrightarrow$ $O(b^d)$ memory size
- Is it complete?
  - tree: no
    graph: yes if space finite
- Is it optimal?
  - No



Memory requirement is a major problem also for GBFS

Outline

A^* Search Strategy

A^* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
  - $g(n)$: cost so far to reach n
  - $h(n)$: estimated cost to goal from n
  - $f(n)$: estimated total cost of path through n to goal

$\Longrightarrow$ Expand first the node n with lowest estimated cost of the cheapest solution through n

- Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A^* Search Strategy

A^* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
  - $g(n)$: cost so far to reach n
  - $h(n)$: estimated cost to goal from n
  - $f(n)$: estimated total cost of path through n to goal

$\Longrightarrow$ Expand first the node n with lowest estimated cost of the cheapest solution through n

- Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A^* Search Strategy

A^* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
  - $g(n)$: cost so far to reach n
  - $h(n)$: estimated cost to goal from n
  - $f(n)$: estimated total cost of path through n to goal

$\Longrightarrow$ Expand first the node n with lowest estimated cost of the cheapest solution through n

- Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A^* Search Strategy

A^* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
  - $g(n)$: cost so far to reach n
  - $h(n)$: estimated cost to goal from n
  - $f(n)$: estimated total cost of path through n to goal

$\Longrightarrow$ Expand first the node n with lowest estimated cost of the cheapest solution through n

- Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A^* Search Strategy

A^* Search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
  - $g(n)$: cost so far to reach n
  - $h(n)$: estimated cost to goal from n
  - $f(n)$: estimated total cost of path through n to goal

$\Longrightarrow$ Expand first the node n with lowest estimated cost of the cheapest solution through n

- Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A^* Search Strategy

A^* Search

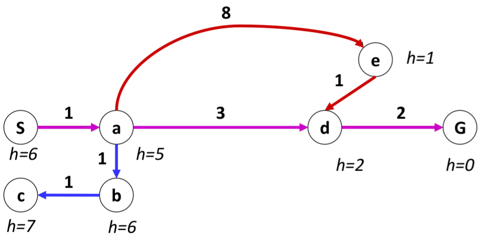
- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Combine Uniform-Cost and Greedy search: $f(n) = g(n) + h(n)$
  - $g(n)$: cost so far to reach n
  - $h(n)$: estimated cost to goal from n
  - $f(n)$: estimated total cost of path through n to goal

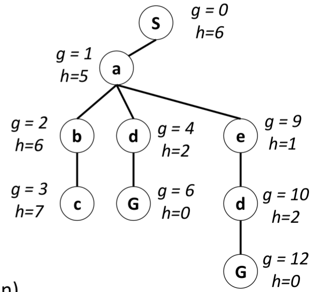
$\Longrightarrow$ Expand first the node n with lowest estimated cost of the cheapest solution through n

- Implementation: same as uniform-cost search, with $g(n) + h(n)$ instead of $g(n)$

A^* Search Strategy: Example

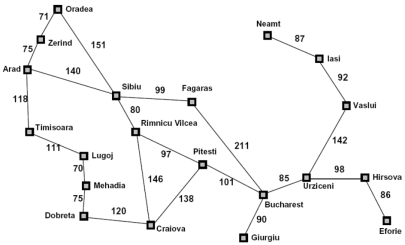
- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)





- A* Search orders by the sum: f(n) = g(n) + h(n)

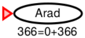
A^* Search Strategy: Example



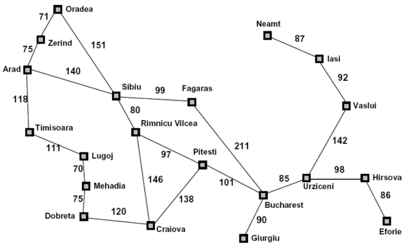
Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

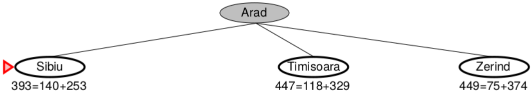


A^* Search Strategy: Example



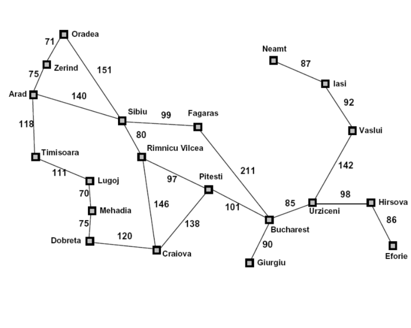
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)



(© S. Russell & P. Norwig, AIMA)

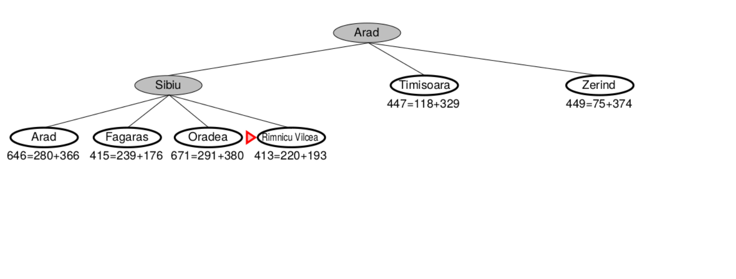
A^* Search Strategy: Example



Straight−line distance to Bucharest

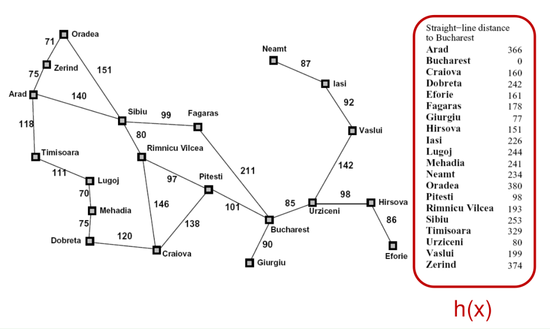
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

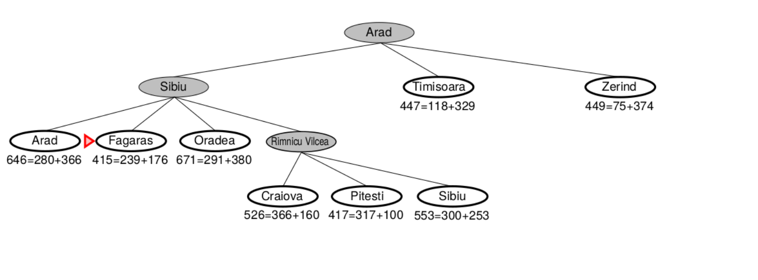
h(x)



(© S. Russell & P. Norwig, AIMA)

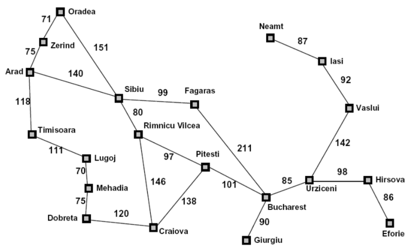
A^* Search Strategy: Example





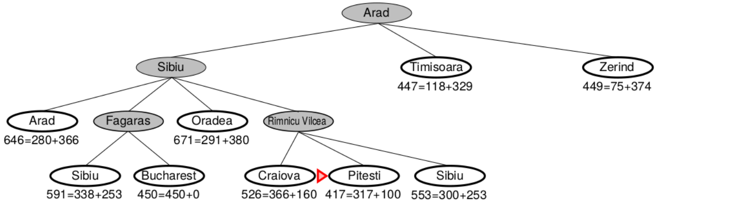
(© S. Russell & P. Norwig, AIMA)

A^* Search Strategy: Example

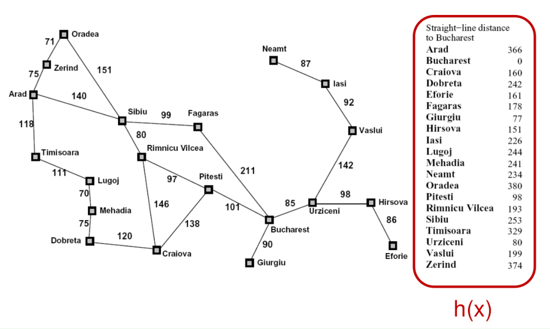


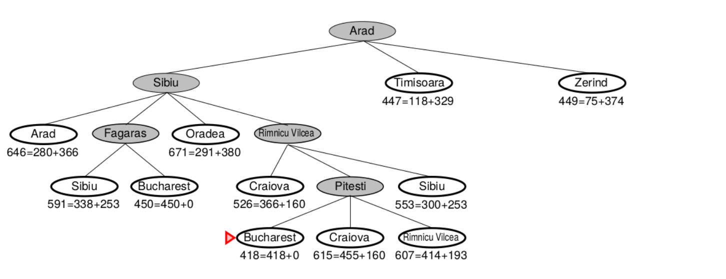
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)



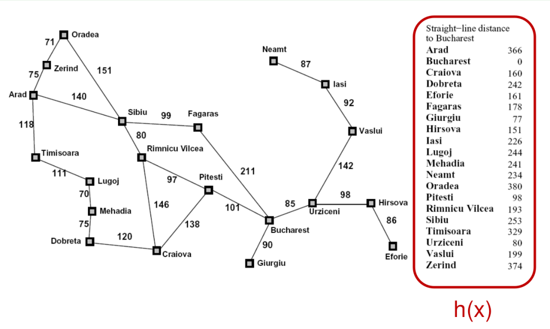
A^* Search Strategy: Example

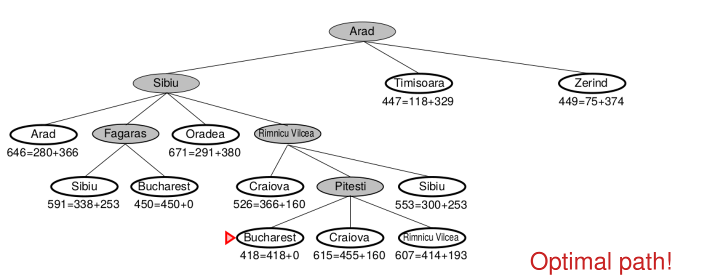




(© S. Russell & P. Norwig, AIMA)

A^* Search Strategy: Example





(© S. Russell & P. Norwig, AIMA)

A^* Search: Admissible and Consistent Heuristics

Admissible heuristics $h(n)$

- $h(n)$ is admissible (aka optimistic) iff it never overestimates the cost to reach the goal:
  - $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n
  - ex: the straight-line distance $h_{SDL}()$ to Bucharest

Consistent heuristics $h(n)$

- $h(n)$ is consistent (aka monotonic) iff, for every successor n' of n generated by any action a with step cost $c(n, a, n')$, $h(n) \leq c(n, a, n') + h(n')$
  - verifies the triangular inequality
  - ex: the straight-line distance $h_{SDL}()$ to Bucharest

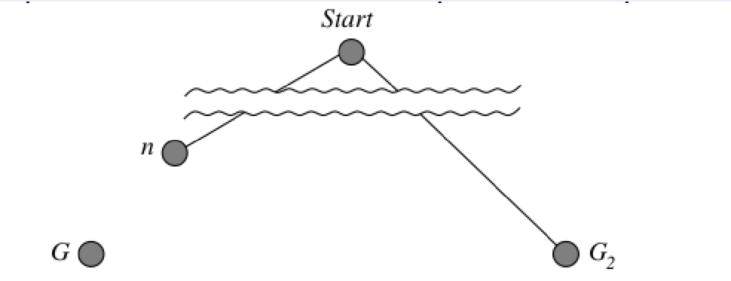
A^* Tree Search: Optimality

If $h(n)$ is admissible, then A^* tree search is optimal

- Suppose some sub-optimal goal G_2 is in the frontier queue.
- n unexpanded node on a shortest path to an optimal goal G.
- then:

$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G) && \text{since } G_2 \text{ sub-optimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

$\Longrightarrow$ A^* will not pick G_2 from the frontier queue before n



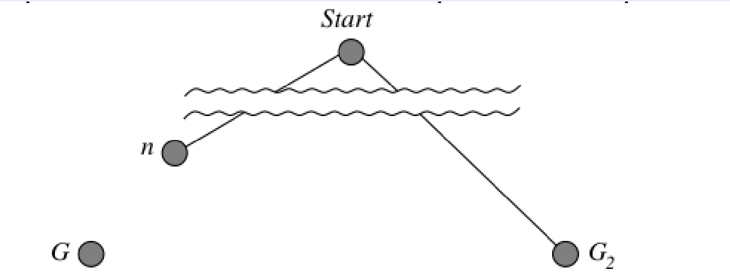
A^* Tree Search: Optimality

If $h(n)$ is admissible, then A^* tree search is optimal

- Suppose some sub-optimal goal G_2 is in the frontier queue.
- n unexpanded node on a shortest path to an optimal goal G.
- then:

$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G) && \text{since } G_2 \text{ sub-optimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

$\Longrightarrow$ A^* will not pick G_2 from the frontier queue before n



(© S. Russell & P. Norwig, AIMA)

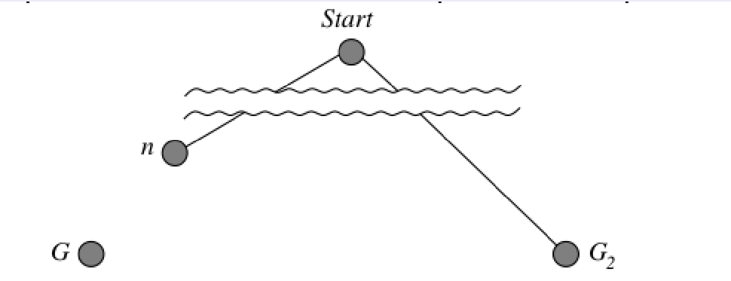
A^* Tree Search: Optimality

If $h(n)$ is admissible, then A^* tree search is optimal

- Suppose some sub-optimal goal G_2 is in the frontier queue.
- n unexpanded node on a shortest path to an optimal goal G.
- then:

$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G) && \text{since } G_2 \text{ sub-optimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

$\Longrightarrow$ A^* will not pick G_2 from the frontier queue before n



(© S. Russell & P. Norwig, AIMA)

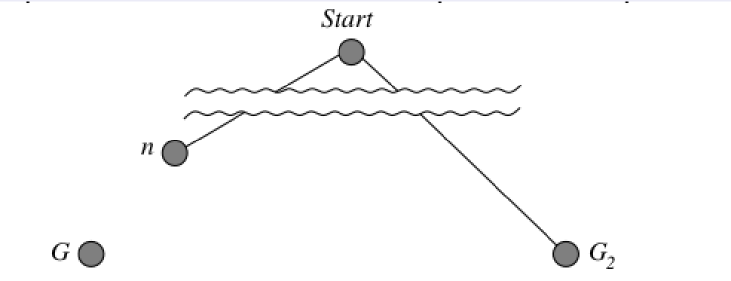
A^* Tree Search: Optimality

If $h(n)$ is admissible, then A^* tree search is optimal

- Suppose some sub-optimal goal G_2 is in the frontier queue.
- n unexpanded node on a shortest path to an optimal goal G.
- then:

$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G) && \text{since } G_2 \text{ sub-optimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

$\Longrightarrow$ A^* will not pick G_2 from the frontier queue before n



(© S. Russell & P. Norwig, AIMA)

A^* Graph Search: Properties

Properties

1. if $h(n)$ is consistent, then $h(n)$ is admissible (straightforward)
2. If h(n) is consistent, then $f(n)$ is non-decreasing along any path:
   - $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$
3. If (Graph) A^* selects a node n from the frontier, then the optimal path to that node has been found
   - if not so, there would be a node n' in the frontier on the optimal path to n (because of the graph separation property)
   - since f is non-decreasing along any path, $f(n') \leq f(n)$
   - since n' is on the optimal path to n, $f(n') < f(n)$

   $\Longrightarrow$ n' would have been selected before n □

$\Longrightarrow$ A^* (graph search) expands nodes in non-decreasing order of f

A^* Graph Search: Properties

Properties

1. if $h(n)$ is consistent, then $h(n)$ is admissible (straightforward)
2. If h(n) is consistent, then $f(n)$ is non-decreasing along any path:
   - $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$
3. If (Graph) A^* selects a node n from the frontier, then the optimal path to that node has been found
   - if not so, there would be a node n' in the frontier on the optimal path to n (because of the graph separation property)
   - since f is non-decreasing along any path, $f(n') \leq f(n)$
   - since n' is on the optimal path to n, $f(n') < f(n)$

   $\Longrightarrow$ n' would have been selected before n □

$\Longrightarrow$ A^* (graph search) expands nodes in non-decreasing order of f

A^* Graph Search: Properties

Properties

1. if $h(n)$ is consistent, then $h(n)$ is admissible (straightforward)
2. If h(n) is consistent, then $f(n)$ is non-decreasing along any path:
   - $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$
3. If (Graph) A^* selects a node n from the frontier, then the optimal path to that node has been found
   - if not so, there would be a node n' in the frontier on the optimal path to n (because of the graph separation property)
   - since f is non-decreasing along any path, $f(n') \leq f(n)$
   - since n' is on the optimal path to n, $f(n') < f(n)$

   $\Longrightarrow$ n' would have been selected before n □

$\Longrightarrow$ A^* (graph search) expands nodes in non-decreasing order of f

A^* Graph Search: Properties

Properties

1. if $h(n)$ is consistent, then $h(n)$ is admissible (straightforward)
2. If h(n) is consistent, then $f(n)$ is non-decreasing along any path:
   - $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$
3. If (Graph) A^* selects a node n from the frontier, then the optimal path to that node has been found
   - if not so, there would be a node n' in the frontier on the optimal path to n (because of the graph separation property)
   - since f is non-decreasing along any path, $f(n') \leq f(n)$
   - since n' is on the optimal path to n, $f(n') < f(n)$

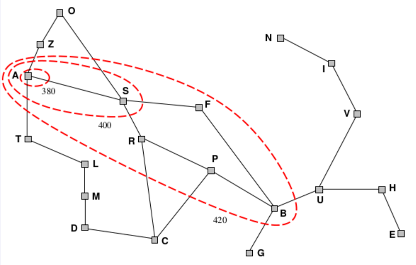
   $\Longrightarrow$ n' would have been selected before n □

$\Longrightarrow$ A^* (graph search) expands nodes in non-decreasing order of f

A^* Graph Search: Optimality

If $h(n)$ is consistent, then A^* graph search is optimal

- A^* expands nodes in order of non-decreasing f value
- Gradually adds "f-contours" of nodes (as BFS adds layers)
  - contour i has all nodes with $f = f_i$, s.t. $f_i < f_{i+1}$
  - cannot expand contour f_{i+1} until contour f_i is fully expanded
- If C^* is the cost of the optimal solution path
  1. A^* expands all nodes s.t. $f(n) < C^*$
  2. A^* might expand some of the nodes on "goal contour" s.t. $f(n) = C^*$ before selecting a goal node.
  3. A^* does not expand nodes s.t. $f(n) > C^*$ (pruning)

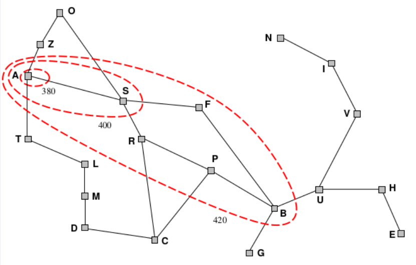


(© S. Russell & P. Norwig, AIMA)

A^* Graph Search: Optimality

If $h(n)$ is consistent, then A^* graph search is optimal

- A^* expands nodes in order of non-decreasing f value
- Gradually adds "f-contours" of nodes (as BFS adds layers)
  - contour i has all nodes with $f = f_i$, s.t. $f_i < f_{i+1}$
  - cannot expand contour f_{i+1} until contour f_i is fully expanded
- If C^* is the cost of the optimal solution path
  1. A^* expands all nodes s.t. $f(n) < C^*$
  2. A^* might expand some of the nodes on "goal contour" s.t. $f(n) = C^*$ before selecting a goal node.
  3. A^* does not expand nodes s.t. $f(n) > C^*$ (pruning)

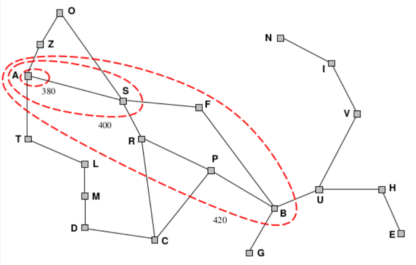


(© S. Russell & P. Norwig, AIMA)

A^* Graph Search: Optimality

If $h(n)$ is consistent, then A^* graph search is optimal

- A^* expands nodes in order of non-decreasing f value
- Gradually adds "f-contours" of nodes (as BFS adds layers)
  - contour i has all nodes with $f = f_i$, s.t. $f_i < f_{i+1}$
  - cannot expand contour f_{i+1} until contour f_i is fully expanded
- If C^* is the cost of the optimal solution path
  1. A^* expands all nodes s.t. $f(n) < C^*$
  2. A^* might expand some of the nodes on "goal contour" s.t. $f(n) = C^*$ before selecting a goal node.
  3. A^* does not expand nodes s.t. $f(n) > C^*$ (pruning)

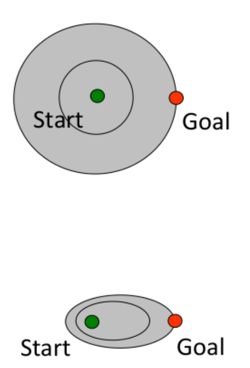


(© S. Russell & P. Norwig, AIMA)

UCS vs A^* Contours

Intuition

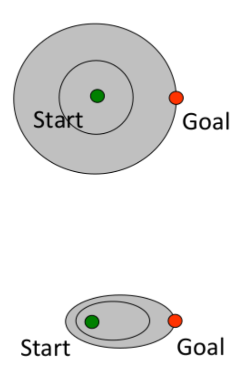
- UCS expands equally in all "directions"
- A^* expands mainly toward the goal



UCS vs A^* Contours

Intuition

- UCS expands equally in all “directions”
- A^* expands mainly toward the goal



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

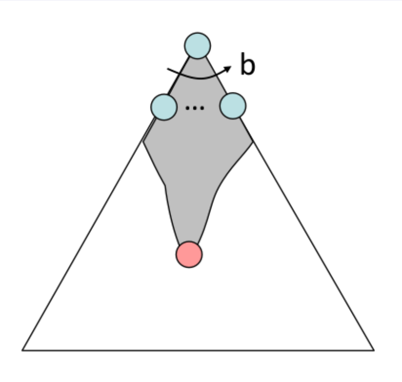
A^* Search: Completeness

If all step costs exceed some finite ϵ and b is finite,
then there are only finitely many nodes n s.t. $f(n) \leq C^*$
$\Longrightarrow$ $A^*$is complete.

A^* Search: Properties

Let $\epsilon \stackrel{\text{def}}{=} (h^+ - h)/h^*$ (relative error)
b^ϵ: effective branching factor

- How many steps?
  - takes $O((b^\epsilon)^d)$ time
    if good heuristics, may give dramatic improvements
- How much memory?
  - Keeps all nodes in memory
    $\Longrightarrow O(b^d)$ memory size
    (like UCS)
- Is it complete?
  - yes
- Is it optimal?
  - yes



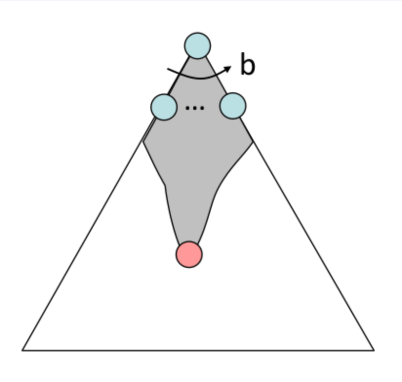
Memory requirement is a major problem also for A^*

A^* Search: Properties

Let $\epsilon \stackrel{\text{def}}{=} (h^+ - h)/h^*$ (relative error)

b^ϵ: effective branching factor

- How many steps?
  - takes $O((b^\epsilon)^d)$ time
    if good heuristics, may give dramatic improvements
- How much memory?
  - Keeps all nodes in memory
    $\Longrightarrow O(b^d)$ memory size
    (like UCS)
- Is it complete?
  - yes
- Is it optimal?
  - yes



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

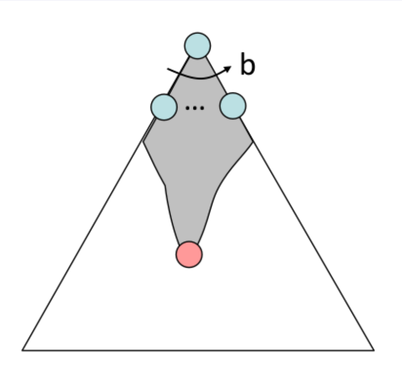
Memory requirement is a major problem also for A^*

A^* Search: Properties

Let $\epsilon \stackrel{\text{def}}{=} (h^+ - h)/h^*$ (relative error)

b^ϵ: effective branching factor

- How many steps?
  - takes $O((b^\epsilon)^d)$ time
    if good heuristics, may give dramatic improvements
- How much memory?
  - Keeps all nodes in memory
    $\Longrightarrow O(b^d)$ memory size
    (like UCS)
- Is it complete?
  - yes
- Is it optimal?
  - yes



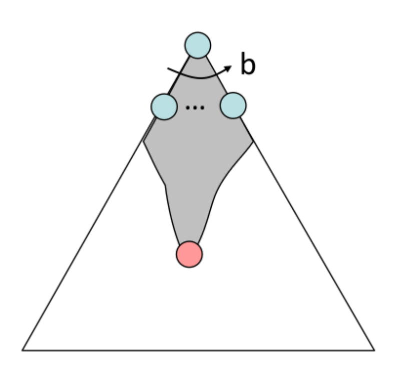
Memory requirement is a major problem also for A^*

A^* Search: Properties

Let $\epsilon \stackrel{\text{def}}{=} (h^+ - h)/h^*$ (relative error)

b^ϵ: effective branching factor

- How many steps?
  - takes $O((b^\epsilon)^d)$ time
    if good heuristics, may give dramatic improvements
- How much memory?
  - Keeps all nodes in memory
    $\Longrightarrow O(b^d)$ memory size
    (like UCS)
- Is it complete?
  - yes
- Is it optimal?
  - yes



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

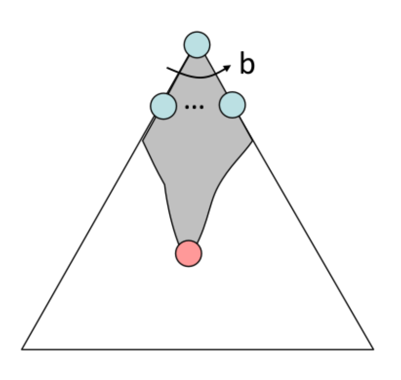
Memory requirement is a major problem also for A^*

A^* Search: Properties

Let $\epsilon \stackrel{\text{def}}{=} (h^+ - h)/h^*$ (relative error)

b^ϵ: effective branching factor

- How many steps?
  - takes $O((b^\epsilon)^d)$ time
    if good heuristics, may give dramatic improvements
- How much memory?
  - Keeps all nodes in memory
    $\Longrightarrow O(b^d)$ memory size
    (like UCS)
- Is it complete?
  - yes
- Is it optimal?
  - yes



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

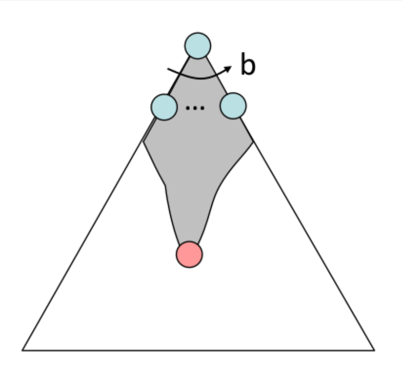
Memory requirement is a major problem also for A^*

A^* Search: Properties

Let $\epsilon \stackrel{\text{def}}{=} (h^+ - h)/h^*$ (relative error)

b^ϵ: effective branching factor

- How many steps?
  - takes $O((b^\epsilon)^d)$ time
    if good heuristics, may give dramatic improvements
- How much memory?
  - Keeps all nodes in memory
    $\Longrightarrow O(b^d)$ memory size
    (like UCS)
- Is it complete?
  - yes
- Is it optimal?
  - yes



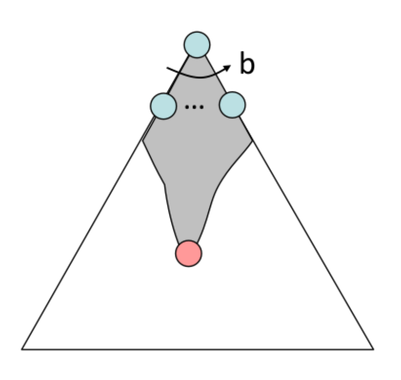
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for A^*

A^* Search: Properties

Let $\epsilon \stackrel{\text{def}}{=} (h^+ - h)/h^*$ (relative error)
b^ϵ: effective branching factor

- How many steps?
  - takes $O((b^\epsilon)^d)$ time
    if good heuristics, may give dramatic improvements
- How much memory?
  - Keeps all nodes in memory
    $\Longrightarrow O(b^d)$ memory size
    (like UCS)
- Is it complete?
  - yes
- Is it optimal?
  - yes



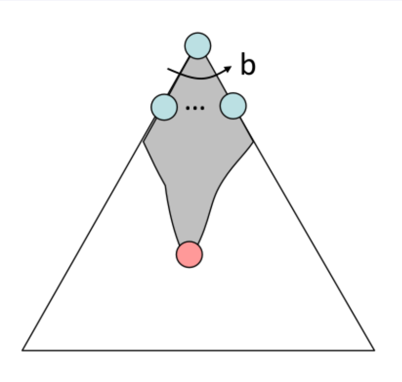
(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Memory requirement is a major problem also for A^*

A^* Search: Properties

Let $\epsilon \stackrel{\text{def}}{=} (h^+ - h)/h^*$ (relative error)
b^ϵ: effective branching factor

- How many steps?
  - takes $O((b^\epsilon)^d)$ time
    if good heuristics, may give dramatic improvements
- How much memory?
  - Keeps all nodes in memory
    $\Longrightarrow O(b^d)$ memory size
    (like UCS)
- Is it complete?
  - yes
- Is it optimal?
  - yes

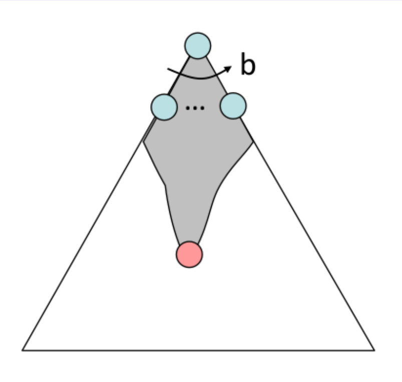


Memory requirement is a major problem also for A^*

A^* Search: Properties

Let $\epsilon \stackrel{\text{def}}{=} (h^+ - h)/h^*$ (relative error)
b^ϵ: effective branching factor

- How many steps?
  - takes $O((b^\epsilon)^d)$ time
    if good heuristics, may give dramatic improvements
- How much memory?
  - Keeps all nodes in memory
    $\Longrightarrow O(b^d)$ memory size
    (like UCS)
- Is it complete?
  - yes
- Is it optimal?
  - yes



Memory requirement is a major problem also for A^*

Memory-bounded Heuristic Search (hints)

Some solutions to A^* space problems
(maintain completeness and optimality)

- Iterative-deepening A^* (IDA^*)
  - here cutoff information is the f-cost (g+h) instead of depth
- Recursive best-first search(RBFS)
  - attempts to mimic standard best-first search with linear space
- (simple) Memory-bounded A^* ((S)MA^*)
  - drop the worst-leaf node when memory is full

Outline

Admissible Heuristics

Main problem

What is the best admissible/consistent heuristic?

Dominance of Admissible Heuristics

Dominance

Let $h_1(n), h_2(n)$ admissible heuristics.

- $h_2(n)$ dominates $h_1(n)$ iff $h_2(n) \geq h_1(n)$ for all n.

$\Longrightarrow$ $h_2(n)$ is better for search

  - is nearer to $h^*(n)$

Let $h_1(n), h_2(n)$ admissible heuristics. Let $h_{12} \stackrel{\text{def}}{=} max(h_1(n), h_2(n))$.

- h_{12} is also admissible
- h_{12} dominates both $h_1(n), h_2(n)$

Dominance of Admissible Heuristics

Dominance

Let $h_1(n), h_2(n)$ admissible heuristics.

- $h_2(n)$ dominates $h_1(n)$ iff $h_2(n) \geq h_1(n)$ for all n.

$\Longrightarrow$ $h_2(n)$ is better for search

- is nearer to $h^*(n)$

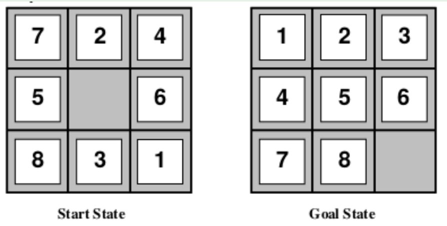
Let $h_1(n), h_2(n)$ admissible heuristics. Let $h_{12} \stackrel{\text{def}}{=} max(h_1(n), h_2(n))$.

- h_{12} is also admissible
- h_{12} dominates both $h_1(n), h_2(n)$

Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
  - (i.e., # of squares from desired location of each tile)
- $h_1(S)$? 6
- $h_2(S)$? 4+0+3+3+1+0+2+1 = 14
- $h^*(S)$? 26
- both $h_1(n)$, $h_2(n)$ admissible ($\leq$ number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$

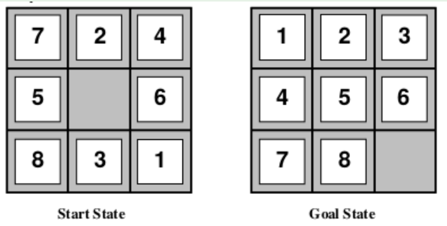


(© S. Russell & P. Norwig, AIMA)

Admissible Heuristics: Example

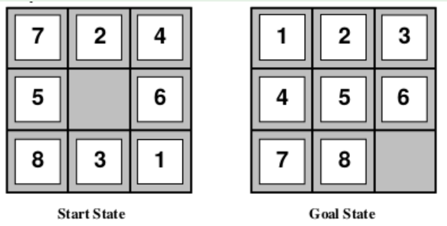
Ex: Heuristics for the 8-puzzle

- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
  - (i.e., # of squares from desired location of each tile)
- $h_1(S)$? 6
- $h_2(S)$? $4+0+3+3+1+0+2+1 = 14$
- $h^*(S)$? 26
- both $h_1(n)$, $h_2(n)$ admissible ($\leq$ number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Ex: Heuristics for the 8-puzzle

- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
  - (i.e., # of squares from desired location of each tile)
- $h_1(S)$? 6
- $h_2(S)$? 4+0+3+3+1+0+2+1 = 14
- $h^*(S)$? 26
- both $h_1(n), h_2(n)$ admissible ($\leq$ number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$

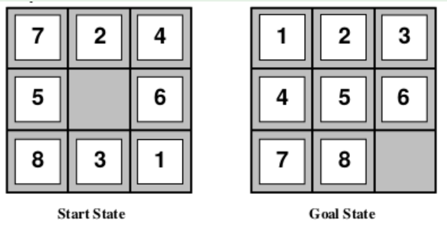


(© S. Russell & P. Norwig, AIMA)

Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
  - (i.e., # of squares from desired location of each tile)
- $h_1(S)$? 6
- $h_2(S)$? 4+0+3+3+1+0+2+1 = 14
- $h^*(S)$? 26
- both $h_1(n)$, $h_2(n)$ admissible ($\leq$ number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$

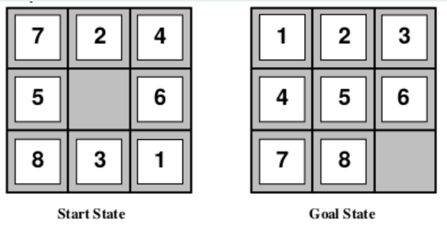


(© S. Russell & P. Norwig, AIMA)

Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

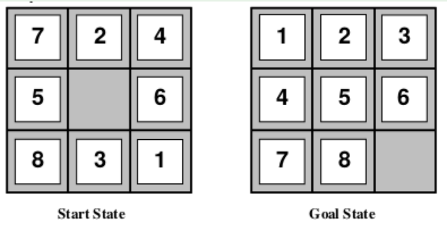
- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
  - (i.e., # of squares from desired location of each tile)
- $h_1(S)$? 6
- $h_2(S)$? 4+0+3+3+1+0+2+1 = 14
- $h^*(S)$? 26
- both $h_1(n)$, $h_2(n)$ admissible ($\leq$ number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
  - (i.e., # of squares from desired location of each tile)
- $h_1(S)$? 6
- $h_2(S)$? 4+0+3+3+1+0+2+1 = 14
- $h^*(S)$? 26
- both $h_1(n)$, $h_2(n)$ admissible ($\leq$ number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$

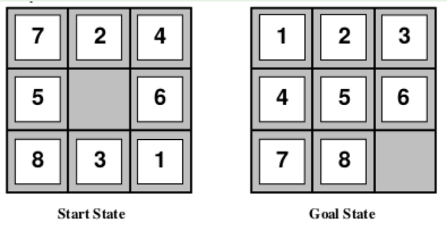


(© S. Russell & P. Norwig, AIMA)

Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

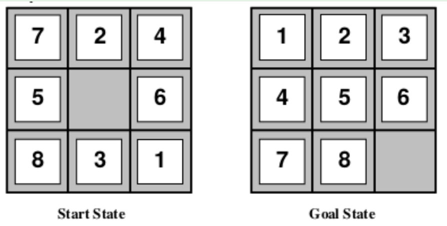
- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
  - (i.e., # of squares from desired location of each tile)
- $h_1(S)$? 6
- $h_2(S)$? 4+0+3+3+1+0+2+1 = 14
- $h^*(S)$? 26
- both $h_1(n)$, $h_2(n)$ admissible ($\leq$ number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Admissible Heuristics: Example

Ex: Heuristics for the 8-puzzle

- $h_1(n)$: number of misplaced tiles
- $h_2(n)$: total Manhattan distance over all tiles
  - (i.e., # of squares from desired location of each tile)
- $h_1(S)$? 6
- $h_2(S)$? 4+0+3+3+1+0+2+1 = 14
- $h^*(S)$? 26
- both $h_1(n)$, $h_2(n)$ admissible ($\leq$ number of actual steps to solve)
- $h_2(n)$ dominates $h_1(n)$



Quality of Heuristics

Effective branching factor

- Effective branching factor b^*: the branching factor that a uniform tree of depth d would have in order to contain N+1 nodes

$$N + 1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d$$

  N being the number of nodes generated by the A^* search
- ex: if d=5 and $N = 52$, then $b^* = 1.92$
- experimental measure of b^* is fairly constant for hard problems
  $\Longrightarrow$ can provide a good guide to the heuristic's overall usefulness
- Ideal value of b^* is 1

Admissible Heuristics: Example [cont.]

Average performances on 100 random samples of 8-puzzle

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

$\Longrightarrow$ Dramatic performance improvement

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
  $\Longrightarrow$ $h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
  $\Longrightarrow$ $h_2(n)$ gives the shortest solution

- The relaxed problem adds edges to the state space
  $\Longrightarrow$ any optimal solution in the original problem is also a solution in the relaxed problem
  $\Longrightarrow$ the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- the derived heuristic is an exact cost for the relaxed problem
  $\Longrightarrow$ must obey the triangular inequality
  $\Longrightarrow$ consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
  $\Longrightarrow$ $h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
  $\Longrightarrow$ $h_2(n)$ gives the shortest solution

- The relaxed problem adds edges to the state space
  $\Longrightarrow$ any optimal solution in the original problem is also a solution in the relaxed problem
  $\Longrightarrow$ the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- the derived heuristic is an exact cost for the relaxed problem
  $\Longrightarrow$ must obey the triangular inequality
  $\Longrightarrow$ consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
  $\Longrightarrow$ $h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
  $\Longrightarrow$ $h_2(n)$ gives the shortest solution

- The relaxed problem adds edges to the state space
  $\Longrightarrow$ any optimal solution in the original problem is also a solution in the relaxed problem
  $\Longrightarrow$ the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- the derived heuristic is an exact cost for the relaxed problem
  $\Longrightarrow$ must obey the triangular inequality
  $\Longrightarrow$ consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
  $\Longrightarrow$ $h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
  $\Longrightarrow$ $h_2(n)$ gives the shortest solution

- The relaxed problem adds edges to the state space
  $\Longrightarrow$ any optimal solution in the original problem is also a solution in the relaxed problem
  $\Longrightarrow$ the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- the derived heuristic is an exact cost for the relaxed problem
  $\Longrightarrow$ must obey the triangular inequality
  $\Longrightarrow$ consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
  $\Longrightarrow$ $h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
  $\Longrightarrow$ $h_2(n)$ gives the shortest solution

- The relaxed problem adds edges to the state space
  $\Longrightarrow$ any optimal solution in the original problem is also a solution in the relaxed problem
  $\Longrightarrow$ the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- the derived heuristic is an exact cost for the relaxed problem
  $\Longrightarrow$ must obey the triangular inequality
  $\Longrightarrow$ consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
  $\Longrightarrow$ $h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
  $\Longrightarrow$ $h_2(n)$ gives the shortest solution

- The relaxed problem adds edges to the state space
  $\Longrightarrow$ any optimal solution in the original problem is also a solution in the relaxed problem
  $\Longrightarrow$ the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- the derived heuristic is an exact cost for the relaxed problem
  $\Longrightarrow$ must obey the triangular inequality
  $\Longrightarrow$ consistent

Admissible Heuristics from Relaxed Problems

Idea: Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

- Relaxed 8-puzzle: a tile can move from any tile to any other tile
  $\Longrightarrow$ $h_1(n)$ gives the shortest solution
- Relaxed 8-puzzle: a tile can move to any adjacent square
  $\Longrightarrow$ $h_2(n)$ gives the shortest solution

- The relaxed problem adds edges to the state space
  $\Longrightarrow$ any optimal solution in the original problem is also a solution in the relaxed problem
  $\Longrightarrow$ the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- the derived heuristic is an exact cost for the relaxed problem
  $\Longrightarrow$ must obey the triangular inequality
  $\Longrightarrow$ consistent

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:
- we can generate three relaxed problems by removing one or both of the conditions

  (*a*) a tile can move from square A to square B if A is adjacent to B

  (*b*) a tile can move from square A to square B if B is blank

  (*c*) a tile can move from square A to square B

$\Longrightarrow$ (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

The tool ABSolver can generate such heuristics automatically.

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:

  a tile can move from square A to square B if
  A is horizontally or vertically adjacent to B, and
  B is blank

- we can generate three relaxed problems by removing one or both of the conditions

  (*a*) a tile can move from square A to square B if A is adjacent to B
  (*b*) a tile can move from square A to square B if B is blank
  (*c*) a tile can move from square A to square B

$\Longrightarrow$ (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

The tool ABSolver can generate such heuristics automatically.

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:

  a tile can move from square A to square B if
  A is horizontally or vertically adjacent to B, and
  B is blank

- we can generate three relaxed problems by removing one or both of the conditions

  (*a*) a tile can move from square A to square B if A is adjacent to B
  (*b*) a tile can move from square A to square B if B is blank
  (*c*) a tile can move from square A to square B

$\Longrightarrow$ (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

The tool ABSolver can generate such heuristics automatically.

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:

  a tile can move from square A to square B if
  A is horizontally or vertically adjacent to B, and
  B is blank

- we can generate three relaxed problems by removing one or both of the conditions

  (*a*) a tile can move from square A to square B if A is adjacent to B

  (*b*) a tile can move from square A to square B if B is blank

  (*c*) a tile can move from square A to square B

$\Longrightarrow$ (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

The tool ABSolver can generate such heuristics automatically.

Inferring Automatically Admissible Heuristics

Idea: If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically

Example

- 8-puzzle actions:

  a tile can move from square A to square B if
  A is horizontally or vertically adjacent to B, and
  B is blank

- we can generate three relaxed problems by removing one or both of the conditions

  (*a*) a tile can move from square A to square B if A is adjacent to B

  (*b*) a tile can move from square A to square B if B is blank

  (*c*) a tile can move from square A to square B

$\Longrightarrow$ (a) corresponds to $h_2(n)$, (c) corresponds to $h_1(n)$,

The tool ABSolver can generate such heuristics automatically.

Learning Admissible Heuristics

- Another way to find an admissible heuristic is through learning from experience:
    - Experience = solving lots of 8-puzzles
    - An inductive learning algorithm can be used to predict costs for other states that arise during search